

# **The Canonical Csound Reference Manual**

**Version 6.09.1**

**Barry Vercoe, MIT Media Lab  
et. al.**

---

# **The Canonical Csound Reference Manual: Version 6.09.1**

by Barry Vercoe and et. al.

---

# Table of Contents

Preface .....	xxxv
Preface to the Csound Manual .....	xxxv
History of the Canonical Csound Reference Manual .....	xxxvi
Copyright Notice .....	xxxvii
Getting Started with Csound .....	xxxix
What's new in Csound 6.09.1 .....	xli
I. Overview .....	1
Introduction .....	4
The Csound Command .....	5
Order of Precedence .....	5
Description of the command syntax .....	5
Csound command line .....	7
Command-line Flags (by Category) .....	17
Csound Environment Variables .....	28
Unified File Format for Orchestras and Scores .....	31
Description .....	31
Example .....	33
Command Line Parameter File (.csoundrc) .....	34
Score File Preprocessing .....	34
The Extract Feature .....	34
Independent Pre-Processing with Scsort .....	35
Using Csound .....	36
Csound's Console Output .....	36
How Csound works .....	37
Amplitude values in Csound .....	38
Real-Time Audio .....	40
Realtime I/O on Linux .....	40
Mac OSX .....	46
Windows .....	47
Realtime I/O with JACK Connection Kit .....	48
Optimizing Audio I/O Latency .....	49
Configuring .....	51
Syntax of the Orchestra .....	52
Orchestra Header Statements .....	52
Instrument and Opcode Block Statements .....	53
Ordinary Statements .....	54
Types, Constants and Variables .....	54
Variable Initialization .....	55
Expressions .....	55
Directories and Files .....	56
Nomenclature .....	56
Macros .....	57
Named Instruments .....	57
User Defined Opcodes (UDO) .....	60
K-Rate Vectors .....	60
Function Syntax in Csound6 .....	60
The Standard Numeric Score .....	62
Preprocessing of Standard Scores .....	62
Carry .....	62
Tempo .....	63
Sort .....	63

Score Statements .....	64
Next-P and Previous-P Symbols .....	64
Ramping .....	65
Score Macros .....	66
Multiple File Score .....	68
Evaluation of Expressions .....	68
Strings in p-fields .....	70
Front Ends .....	71
CsoundAC .....	72
CsoundVST .....	74
Building Csound .....	76
Csound Links .....	77
II. Opcodes Overview .....	78
Signal Generators .....	82
Additive Synthesis/Resynthesis .....	82
Basic Oscillators .....	82
Dynamic Spectrum Oscillators .....	82
FM Synthesis .....	83
Granular Synthesis .....	83
Hyper Vectorial Synthesis .....	84
Linear and Exponential Generators .....	84
Envelope Generators .....	85
Models and Emulations .....	85
Phasors .....	86
Random (Noise) Generators .....	87
Sample Playback .....	88
Soundfonts .....	88
Scanned Synthesis .....	90
Table Access .....	91
Wave Terrain Synthesis .....	92
Waveguide Physical Modeling .....	92
Signal Input and Output .....	93
File Input and Output .....	93
Signal Input .....	93
Signal Output .....	93
Software Bus .....	94
Printing and Display .....	94
Sound File Queries .....	94
Signal Modifiers .....	96
Amplitude Modifiers and Dynamic processing .....	96
Convolution and Morphing .....	96
Delay .....	96
Panning and Spatialization .....	97
Reverberation .....	98
Sample Level Operators .....	99
Signal Limiters .....	99
Special Effects .....	99
Standard Filters .....	100
Specialized Filters .....	102
Waveguides .....	102
Waveshaping and Phase Distortion .....	102
Instrument Control .....	104
Clock Control .....	104
Conditional Values .....	104

Duration Control Statements .....	104
FLTK Widgets and GUI controllers .....	104
FLTK Containers .....	107
FLTK Valuators .....	107
Other FLTK Widgets .....	108
Modifying FLTK Widget Appearance .....	108
General FLTK Widget-related Opcodes .....	109
Instrument Invocation .....	109
Program Flow Control .....	110
Real-time Performance Control .....	111
Initialization and Reinitialization .....	111
Sensing and Control .....	112
Stacks .....	113
Sub-instrument Control .....	113
Time Reading .....	113
Function Table Control .....	115
Table Queries .....	115
Read/Write Operations .....	115
Table Reading with Dynamic Selection .....	116
Mathematical Operations .....	117
Amplitude Converters .....	117
Arithmetic and Logic Operations .....	117
Comparators and Accumulators .....	117
Mathematical Functions .....	118
Opcode Equivalents of Functions .....	118
Random Functions .....	118
Trigonometric Functions .....	119
Linear Algebra Opcodes .....	120
Array Opcodes .....	128
Pitch Converters .....	135
Functions .....	135
Tuning Opcodes .....	135
Real-time MIDI Support .....	136
Virtual MIDI Keyboard .....	137
MIDI input .....	140
MIDI Message Output .....	140
Generic Input and Output .....	141
Converters .....	141
Event Extenders .....	141
Note-on/Note-off Output .....	141
MIDI/Score Interoperability opcodes .....	142
System Realtime Messages .....	143
Slider Banks .....	143
Spectral Processing .....	144
Short-time Fourier Transform (STFT) Resynthesis .....	144
Linear Predictive Coding (LPC) Resynthesis .....	144
Non-standard Spectral Processing .....	145
Tools for Real-time Spectral Processing (pvs opcodes) .....	145
ATS Spectral Processing .....	146
Loris Opcodes .....	147
Array-based spectral opcodes .....	150
Strings .....	152
String Manipulation Opcodes .....	153
String Conversion Opcodes .....	154

Vectorial Opcodes .....	155
Tables of vectors operators .....	155
Operations Between a Vectorial and a Scalar Signal .....	155
Operations Between two Vectorial Signals .....	156
Vectorial Envelope Generators .....	156
Limiting and wrapping of vectorial control signals .....	157
Vectorial Control-rate Delay Paths .....	157
Vectorial Random Signal Generators .....	157
Zak Patch System .....	158
Plugin Hosting .....	159
DSSI and LADSPA for Csound .....	159
VST for Csound .....	159
OSC and Network .....	161
Ableton Link Opcodes .....	161
OSC .....	161
Network .....	162
Remote Opcodes .....	162
Mixer Opcodes .....	163
Signal Flow Graph Opcodes .....	164
Jacko Opcodes .....	167
Lua Opcodes .....	170
Python Opcodes .....	175
Introduction .....	175
Orchestra Syntax .....	175
Image processing opcodes .....	177
STK Opcodes .....	178
Miscellaneous opcodes .....	180
III. Reference .....	181
Orchestra Opcodes and Operators .....	210
!= .....	211
#define .....	213
#include .....	217
#undef .....	219
#ifdef .....	220
#ifndef .....	221
\$NAME .....	222
% .....	225
&& .....	227
> .....	229
>= .....	231
< .....	233
<= .....	235
* .....	237
+ .....	240
- .....	243
/ .....	246
= .....	249
+= .....	251
== .....	253
^ .....	255
.....	257
Odbfs .....	260
A4 .....	263
<< .....	264

The Canonical Csound  
Reference Manual

---

>>	266
&	267
	269
¬	270
#	271
a	272
abs	274
active	276
adsr	280
adsyn	283
adsynt	285
adsynt2	288
aftouch	291
alpass	293
alwayson	295
ampdb	298
ampdbfs	300
ampmidi	302
ampmidid	304
areson	306
aresonk	308
atone	310
atonek	312
atonex	314
ATSadd	316
ATSaddnz	320
ATSbufread	323
ATScross	325
ATSinfo	328
ATSinterpread	331
ATSread	333
ATSreadnz	337
ATSpartialtap	340
ATSSinnoi	342
babo	346
balance	350
bamboo	352
barmodel	354
bbcutm	356
bbcuts	361
betarand	364
bexprnd	367
bformenc1	369
bformdec1	372
binit	375
biquad	377
biquada	382
birnd	385
bpf	387
bqrez	389
butbp	391
butbr	392
buthp	393
butlp	394

The Canonical Csound  
Reference Manual

---

butterbp .....	395
butterbr .....	397
butterhp .....	399
butterlp .....	401
button .....	403
buzz .....	405
c2r .....	407
cabasa .....	409
cauchy .....	411
cauchy1 .....	413
ceil .....	415
cell .....	417
cent .....	420
centroid .....	422
ceps .....	424
cepsinv .....	426
cggoto .....	428
chanctrl .....	430
changed .....	432
changed2 .....	434
chani .....	437
chano .....	438
cbrt .....	439
chebyshevpoly .....	441
checkbox .....	444
chn .....	446
chnclear .....	448
chnexport .....	450
chnget .....	452
chnmix .....	455
chnparams .....	457
chnset .....	458
chuap .....	461
cigoto .....	465
ckgoto .....	467
clear .....	469
clfilt .....	471
clip .....	474
clockoff .....	476
clockon .....	478
cmp .....	480
cmplxprod .....	482
cngoto .....	484
comb .....	486
combinv .....	488
compilecsd .....	490
compileorc .....	492
compilestr .....	494
compress .....	496
compress2 .....	499
connect .....	502
control .....	505
convle .....	506
convolve .....	507

The Canonical Csound  
Reference Manual

---

copya2ftab .....	511
copyf2array .....	513
cos .....	515
cosseg .....	517
cossegb .....	519
cossegr .....	521
cosh .....	523
cosinv .....	525
cps2pch .....	527
cpsmidi .....	531
cpsmidib .....	533
cpsmidinn .....	535
cpsoct .....	539
cpspch .....	542
cpstmid .....	545
cpstun .....	547
cpstuni .....	550
cpsxpch .....	553
cpumeter .....	557
cpuprc .....	559
cross2 .....	562
crossfm .....	564
crunch .....	567
ctrl14 .....	569
ctrl21 .....	571
ctrl7 .....	573
ctrlinit .....	576
cuserrnd .....	577
dam .....	580
date .....	583
dates .....	585
db .....	587
dbamp .....	589
dbfsamp .....	591
dcblock .....	593
dcblock2 .....	595
dconv .....	597
dct .....	599
dctinv .....	601
delay .....	603
delay1 .....	605
delayk .....	607
delayr .....	610
delayw .....	612
deltap .....	614
deltap3 .....	617
deltapi .....	620
deltapn .....	623
deltapx .....	625
deltapxw .....	627
denorm .....	629
diff .....	631
diode_ladder .....	633
directory .....	636

The Canonical Csound  
Reference Manual

---

diskgrain .....	638
diskin .....	641
diskin2 .....	644
dispfft .....	648
display .....	650
distort .....	652
distort1 .....	654
divz .....	656
doppler .....	658
dot .....	660
downsamp .....	661
dripwater .....	663
dssiactivate .....	665
dssiaudio .....	668
dssictls .....	670
dssiinit .....	672
dssilist .....	674
dumpk .....	676
dumpk2 .....	679
dumpk3 .....	682
dumpk4 .....	685
duserrnd .....	688
dust .....	690
dust2 .....	692
else .....	694
elseif .....	696
endif .....	698
endin .....	700
endop .....	702
envlpx .....	705
envlpxr .....	708
ephasor .....	711
eqfil .....	713
evalstr .....	715
event .....	716
event_i .....	720
exciter .....	722
exitnow .....	724
exp .....	726
expcurve .....	728
expon .....	730
exprand .....	732
exprandi .....	734
expseg .....	736
expsega .....	738
expsegb .....	740
expsegba .....	742
expsegr .....	744
faustaudio .....	746
faustcompile .....	747
faustctl .....	748
faustgen .....	749
fareylen .....	751
fareyleni .....	753

ficlose .....	756
filebit .....	758
filelen .....	760
filenchnls .....	762
filepeak .....	764
filesca1 .....	766
filesr .....	768
filevalid .....	770
fillarray .....	772
fft .....	774
filter2 .....	776
fin .....	778
fini .....	780
fink .....	782
fiopen .....	784
flanger .....	786
flashtxt .....	788
FLbox .....	790
FLbutBank .....	795
FLbutton .....	798
FLcloseButton .....	803
FLcolor .....	806
FLcolor2 .....	808
FLcount .....	809
FLexecButton .....	812
FLgetsnap .....	815
FLgroup .....	816
FLgroupEnd .....	818
FLgroup_end .....	819
FLhide .....	820
FLhvsBox .....	821
FLhvsBoxSetValue .....	822
FLjoy .....	823
FLkeyIn .....	827
FLknob .....	829
FLlabel .....	834
FLloadsnap .....	836
FLmouse .....	837
flooper .....	839
flooper2 .....	841
floor .....	843
FLpack .....	845
FLpackEnd .....	848
FLpack_end .....	849
FLpanel .....	850
FLpanelEnd .....	854
FLpanel_end .....	855
FLprintk .....	856
FLprintk2 .....	857
FLroller .....	858
FLrun .....	861
FLsavesnap .....	862
FLscroll .....	868
FLscrollEnd .....	871

FLscroll_end .....	872
FLsetAlign .....	873
FLsetBox .....	874
FLsetColor .....	876
FLsetColor2 .....	878
FLsetFont .....	879
FLsetPosition .....	881
FLsetSize .....	882
FLsetsnap .....	883
FLsetSnapGroup .....	885
FLsetText .....	886
FLsetTextColor .....	888
FLsetTextSize .....	889
FLsetTextType .....	890
FLsetVal_i .....	893
FLsetVal .....	894
FLshow .....	895
FLslidBnk .....	896
FLslidBnk2 .....	900
FLslidBnkGetHandle .....	903
FLslidBnkSet .....	904
FLslidBnkSetk .....	905
FLslidBnk2Set .....	907
FLslidBnk2Setk .....	908
FLslider .....	911
FLtabs .....	917
FLtabsEnd .....	923
FLtabs_end .....	924
FLtext .....	925
FLupdate .....	928
fluidAllOut .....	929
fluidCCi .....	931
fluidCCk .....	933
fluidControl .....	935
fluidEngine .....	938
fluidLoad .....	941
fluidNote .....	943
fluidOut .....	945
fluidProgramSelect .....	948
fluidSetInterpMethod .....	951
FLvalue .....	953
FLvkeybd .....	956
FLvslidBnk .....	957
FLvslidBnk2 .....	961
FLxyin .....	963
fmanal .....	966
fmax .....	968
fmb3 .....	970
fmbell .....	972
fmin .....	975
fmmetal .....	977
fmod .....	979
fmpercfl .....	981
fmrhode .....	983

The Canonical Csound  
Reference Manual

---

fmvoice .....	985
fmwurlie .....	987
fof .....	989
fof2 .....	992
fofilter .....	998
fog .....	1000
fold .....	1003
follow .....	1005
follow2 .....	1007
foscil .....	1009
foscili .....	1011
fout .....	1013
fouti .....	1017
foutir .....	1019
foutk .....	1021
fprintks .....	1023
fprints .....	1029
frac .....	1031
fractalnoise .....	1033
framebuffer .....	1035
freeverb .....	1037
ftchnls .....	1039
ftconv .....	1041
ftcps .....	1044
ftfree .....	1046
ftgen .....	1048
ftgenonce .....	1051
ftgentmp .....	1053
ftlen .....	1055
ftload .....	1057
ftloadk .....	1058
ftlptim .....	1059
ftmorf .....	1061
ftom .....	1063
ftsamplbank .....	1065
ftsav .....	1067
ftsavk .....	1069
ftsr .....	1070
gain .....	1072
gainslider .....	1074
gauss .....	1076
gaussi .....	1078
gausstrig .....	1080
gbuzz .....	1083
genarray .....	1085
genarray_i .....	1087
gendy .....	1089
gendyc .....	1093
gendyx .....	1096
getcfg .....	1100
getcol .....	1102
getftargs .....	1104
getrow .....	1106
getseed .....	1108

The Canonical Csound  
Reference Manual

---

gogobel .....	1109
goto .....	1111
grain .....	1113
grain2 .....	1115
grain3 .....	1120
granule .....	1125
guiro .....	1128
harmon .....	1130
harmon2 .....	1132
hdf5read .....	1135
hdf5write .....	1137
hilbert .....	1139
hilbert2 .....	1144
hrtfearly .....	1146
hrtfmove .....	1150
hrtfmove2 .....	1153
hrtfreverb .....	1156
hrtfstat .....	1159
hsboscil .....	1162
hvs1 .....	1165
hvs2 .....	1169
hvs3 .....	1175
hypot .....	1178
i .....	1180
if .....	1181
fftnv .....	1186
igoto .....	1188
ihold .....	1190
imagecreate .....	1192
imagefree .....	1194
imagegetpixel .....	1196
imageload .....	1199
imagesave .....	1201
imagesetpixel .....	1203
imagesize .....	1205
in .....	1207
in32 .....	1209
inch .....	1210
inh .....	1212
init .....	1213
initc14 .....	1216
initc21 .....	1217
initc7 .....	1218
inleta .....	1220
inletk .....	1223
inletkid .....	1225
inletf .....	1226
inletv .....	1227
ino .....	1228
inq .....	1229
inrg .....	1231
ins .....	1232
insremot .....	1234
insglobal .....	1237

The Canonical Csound  
Reference Manual

---

instr .....	1239
int .....	1241
integ .....	1243
interp .....	1245
invalue .....	1248
inx .....	1250
inz .....	1251
JackoAudioIn .....	1252
JackoAudioInConnect .....	1253
JackoAudioOut .....	1254
JackoAudioOutConnect .....	1255
JackoFreewheel .....	1256
JackoInfo .....	1257
JackoInit .....	1259
JackoMidiInConnect .....	1261
JackoMidiOutConnect .....	1262
JackoMidiOut .....	1263
JackoNoteOut .....	1264
JackoOn .....	1265
JackoTransport .....	1266
jacktransport .....	1267
jitter .....	1269
jitter2 .....	1271
joystick .....	1273
jspline .....	1276
k .....	1278
K35_hpf .....	1279
K35_lpf .....	1283
kgoto .....	1287
kr .....	1289
ksmps .....	1290
lenarray .....	1291
lfo .....	1293
limit .....	1295
limit1 .....	1297
line .....	1299
linen .....	1301
linenr .....	1303
lineto .....	1306
linlin .....	1308
link_beat_force .....	1310
link_beat_get .....	1311
link_beat_request .....	1312
link_create .....	1313
link_enable .....	1315
link_is_enabled .....	1317
link_metro .....	1319
link_peers .....	1321
link_tempo_get .....	1323
link_tempo_set .....	1325
linrand .....	1327
linseg .....	1329
linsegb .....	1331
linsegr .....	1333

The Canonical Csound  
Reference Manual

---

liveconv .....	1335
locsend .....	1338
locsig .....	1341
log .....	1344
log10 .....	1346
log2 .....	1348
logbtwo .....	1350
logcurve .....	1352
loop_ge .....	1354
loop_gt .....	1356
loop_le .....	1358
loop_lt .....	1361
loopseg .....	1364
loopsegp .....	1366
looptseg .....	1368
loopxseg .....	1370
lorenz .....	1372
lorisread .....	1375
lorismorph .....	1378
lorisplay .....	1381
loscil .....	1384
loscil3 .....	1387
loscilx .....	1390
lowpass2 .....	1393
lowres .....	1395
lowresx .....	1397
lpf18 .....	1399
lpfreson .....	1401
lphasor .....	1403
lpinterp .....	1405
lposcil .....	1406
lposcil3 .....	1408
lposcila .....	1410
lposcilsa .....	1412
lposcilsa2 .....	1414
lpread .....	1416
lpreson .....	1419
lpshold .....	1422
lpsholdp .....	1424
lpslot .....	1425
lua_exec .....	1426
lua_opdef .....	1427
lua_opcall .....	1433
mac .....	1436
maca .....	1438
madsr .....	1440
mags .....	1444
mandel .....	1447
mandol .....	1450
maparray .....	1452
marimba .....	1455
massign .....	1458
max .....	1461
maxabs .....	1463

The Canonical Csound  
Reference Manual

---

maxabsaccum .....	1465
maxaccum .....	1467
maxalloc .....	1469
max_k .....	1471
maxarray .....	1473
mclock .....	1475
mdelay .....	1477
median .....	1479
mediank .....	1481
metro .....	1483
mfb .....	1485
midglobal .....	1487
midiarp .....	1488
midic14 .....	1490
midic21 .....	1492
midic7 .....	1494
midichannelaftertouch .....	1496
midichn .....	1498
midicontrolchange .....	1501
midictrl .....	1503
mididefault .....	1505
midiin .....	1507
midifilestatus .....	1510
midinoteoff .....	1511
midinoteoncps .....	1513
midinoteonkey .....	1515
midinoteonoct .....	1517
midinoteonpch .....	1519
midion2 .....	1521
midion .....	1523
midiont .....	1526
midipitchbend .....	1528
midipolyaftertouch .....	1530
midiprogramchange .....	1533
miditempo .....	1535
midremot .....	1537
min .....	1540
minabs .....	1542
minabsaccum .....	1544
minaccum .....	1546
mincer .....	1548
minarray .....	1550
mirror .....	1552
MixerSetLevel .....	1554
MixerSetLevel_i .....	1557
MixerGetLevel .....	1558
MixerSend .....	1560
MixerReceive .....	1562
MixerClear .....	1564
mode .....	1566
modmatrix .....	1569
monitor .....	1574
moog .....	1576
moogladder .....	1578

moogladder2 .....	1580
moogvcf .....	1582
moogvcf2 .....	1584
moscil .....	1586
mp3in .....	1588
mp3len .....	1590
mp3scal .....	1592
mpulse .....	1594
mrtmsg .....	1596
mtof .....	1597
mton .....	1599
multitap .....	1601
mute .....	1603
mvchpf .....	1605
mvclpf1 .....	1607
mvclpf2 .....	1609
mvclpf3 .....	1611
mvclpf4 .....	1613
mxadsr .....	1615
nchnls .....	1618
nchnls_hw .....	1620
nchnls_i .....	1621
nestedap .....	1623
nlfilt .....	1626
nlfilt2 .....	1629
noise .....	1632
noteoff .....	1635
noteon .....	1636
noteondur2 .....	1637
noteondur .....	1639
notnum .....	1641
nreverb .....	1643
nrpn .....	1646
nsamp .....	1648
nstance .....	1650
ntom .....	1653
nstrnum .....	1655
ntrpol .....	1656
octave .....	1658
octcps .....	1660
octmidi .....	1663
octmidib .....	1665
octmidinn .....	1667
octpch .....	1670
olabuffer .....	1673
opcode .....	1675
oscbnk .....	1680
oscil1 .....	1686
oscil1i .....	1688
oscil3 .....	1690
oscil .....	1692
oscili .....	1694
oscilikt .....	1696
osciliktp .....	1698

The Canonical Csound  
Reference Manual

---

oscilikts .....	1700
osciln .....	1702
oscils .....	1704
oscilx .....	1706
OSCinit .....	1707
OSClisten .....	1709
OSCrow .....	1713
OSCsend .....	1715
out32 .....	1717
out .....	1718
outc .....	1720
outch .....	1722
outh .....	1725
outiat .....	1726
outic14 .....	1728
outic .....	1729
outipat .....	1731
outipb .....	1732
outipc .....	1734
outkat .....	1736
outkc14 .....	1738
outkc .....	1739
outkpat .....	1741
outkpb .....	1742
outkpc .....	1744
outleta .....	1747
outletf .....	1749
outletk .....	1750
outletkid .....	1752
outletv .....	1753
outo .....	1754
outq1 .....	1755
outq2 .....	1757
outq3 .....	1759
outq4 .....	1761
outq .....	1763
outrg .....	1765
outs1 .....	1767
outs2 .....	1769
outs .....	1771
outvalue .....	1773
outx .....	1775
outz .....	1776
p5gconnect .....	1777
p5gdata .....	1779
p .....	1781
pan2 .....	1783
pan .....	1785
pareq .....	1787
partials .....	1790
partikkel .....	1792
partikkelget .....	1801
partikkelset .....	1804
partikkelsync .....	1807

The Canonical Csound  
Reference Manual

---

passign .....	1811
paulstretch .....	1813
pcauchy .....	1815
pchbend .....	1817
pchmidi .....	1819
pchmidib .....	1821
pchmidinn .....	1823
pchoct .....	1826
pctom .....	1829
pconvolve .....	1831
pcount .....	1834
pdclip .....	1837
pdhalf .....	1840
pdhalfy .....	1843
peak .....	1846
pgmassign .....	1848
phaser1 .....	1852
phaser2 .....	1855
phasor .....	1859
phasorbnk .....	1861
phs .....	1863
pindex .....	1866
pinker .....	1870
pinkish .....	1871
pitch .....	1874
pitchamdf .....	1877
planet .....	1880
platerev .....	1883
plltrack .....	1885
pluck .....	1887
poisson .....	1889
pol2rect .....	1893
polyaft .....	1896
polynomial .....	1898
port .....	1901
portk .....	1903
poscil3 .....	1905
poscil .....	1908
pow .....	1910
powershape .....	1912
powoftwo .....	1914
prealloc .....	1916
prepiano .....	1918
print .....	1921
printf .....	1923
printk2 .....	1925
printk .....	1927
printks .....	1929
printks2 .....	1932
prints .....	1934
product .....	1936
product .....	1938
pset .....	1939
ptable .....	1941

The Canonical Csound  
Reference Manual

---

ptablei .....	1943
ptable3 .....	1946
ptablew .....	1947
ptrack .....	1950
puts .....	1952
pvadd .....	1954
pvbufread .....	1958
pvcross .....	1960
pvinterp .....	1963
pvoc .....	1966
pvread .....	1968
pvsadsyn .....	1970
pvsanal .....	1972
pvsarp .....	1975
pvsbandp .....	1978
pvsbandr .....	1980
pvsbin .....	1982
pvsblur .....	1984
pvsbuffer .....	1986
pvsbufread .....	1987
pvsbufread2 .....	1990
pvscale .....	1992
pvscent .....	1994
pvsceps .....	1996
pvcross .....	1998
pvsdemix .....	2000
pvsdiskin .....	2002
pvsdisp .....	2004
pvsfilter .....	2006
pvsfread .....	2009
pvsfreeze .....	2011
pvsftr .....	2013
pvsftw .....	2015
pvsfwrite .....	2017
pvsgain .....	2019
pvshift .....	2021
pvsifd .....	2023
pvsinfo .....	2025
pvsinit .....	2027
pvsin .....	2028
pvslock .....	2029
pvsmaska .....	2031
pvsmix .....	2033
pvsmorph .....	2035
pvssmooth .....	2038
pvsout .....	2040
pvsosc .....	2041
pvspitch .....	2044
pvstanal .....	2047
pvstencil .....	2049
pvstrace .....	2051
pvsvoc .....	2053
pvsynth .....	2055
pvs warp .....	2057

pvs2tab .....	2059
pyassign Opcodes .....	2060
pycall Opcodes .....	2061
pyeval Opcodes .....	2065
pyexec Opcodes .....	2066
pyinit Opcodes .....	2069
pyrun Opcodes .....	2070
pwd .....	2072
qinf .....	2074
qnan .....	2076
r2c .....	2078
rand .....	2080
randh .....	2082
randi .....	2084
random .....	2086
randomh .....	2088
randomi .....	2091
rbjeq .....	2094
readclock .....	2097
readf .....	2099
readfi .....	2101
readk .....	2103
readk2 .....	2106
readk3 .....	2109
readk4 .....	2112
readscore .....	2115
readscratch .....	2117
rect2pol .....	2119
reinit .....	2121
release .....	2123
remoteport .....	2124
remove .....	2125
repluck .....	2126
reson .....	2128
resonk .....	2130
resonr .....	2132
resonx .....	2135
resonxk .....	2137
resony .....	2139
resonz .....	2141
resyn .....	2144
return .....	2146
reverb .....	2147
reverb2 .....	2149
reverb3 .....	2150
rewindscore .....	2152
rezzy .....	2154
rfft .....	2156
rifft .....	2158
rigoto .....	2160
rireturn .....	2161
rms .....	2163
rnd .....	2165
rnd31 .....	2167

round .....	2173
rspline .....	2175
rtclock .....	2177
S .....	2179
s16b14 .....	2180
s32b14 .....	2182
samphold .....	2184
sandpaper .....	2186
scale .....	2188
scalearray .....	2190
scanhammer .....	2192
scans .....	2193
scantable .....	2197
scanu .....	2199
schedkwhen .....	2201
schedkwhennamed .....	2204
schedule .....	2206
schedwhen .....	2209
scoreline .....	2211
scoreline_i .....	2213
sc_lag .....	2215
sc_lagud .....	2217
sc_phasor .....	2219
sc_trig .....	2221
seed .....	2223
sekere .....	2225
select .....	2227
semitone .....	2229
sense .....	2231
sensekey .....	2232
serialBegin .....	2236
serialEnd .....	2238
serialFlush .....	2239
serialPrint .....	2240
serialRead .....	2241
serialWrite_i .....	2243
serialWrite .....	2244
seqtime2 .....	2246
seqtime .....	2249
setctrl .....	2252
setksmps .....	2255
setrow .....	2257
setscorepos .....	2259
sfilist .....	2261
sfinstr3 .....	2263
sfinstr3m .....	2266
sfinstr .....	2269
sfinstrm .....	2272
sfload .....	2274
sflooper .....	2277
sfpassign .....	2280
sfplay3 .....	2283
sfplay3m .....	2286
sfplay .....	2289

sfplaym .....	2291
sfplist .....	2294
sfpreset .....	2296
shaker .....	2299
shiftin .....	2301
shiftout .....	2303
signum .....	2305
sin .....	2307
sinh .....	2309
sininv .....	2311
sinsyn .....	2313
sleighbells .....	2315
slicearray .....	2317
slider16 .....	2319
slider16f .....	2321
slider16table .....	2323
slider16tablef .....	2325
slider32 .....	2327
slider32f .....	2329
slider32table .....	2331
slider32tablef .....	2333
slider64 .....	2335
slider64f .....	2337
slider64table .....	2339
slider64tablef .....	2341
slider8 .....	2343
slider8f .....	2345
slider8table .....	2347
slider8tablef .....	2349
sliderKawai .....	2351
sndloop .....	2352
sndwarp .....	2354
sndwarpst .....	2358
sockrecv .....	2362
socksend .....	2364
sorta .....	2366
sortd .....	2367
soundin .....	2368
space .....	2371
spat3d .....	2376
spat3di .....	2385
spat3dt .....	2389
spdist .....	2394
specaddm .....	2398
specdiff .....	2399
specdisp .....	2400
specfilt .....	2401
spechist .....	2402
specptrk .....	2403
specscal .....	2405
specsum .....	2406
spectrum .....	2407
splitrig .....	2409
sprintf .....	2411

The Canonical Csound  
Reference Manual

---

sprintfk .....	2413
spsend .....	2415
sqrt .....	2418
sr .....	2420
statevar .....	2422
stix .....	2424
STKBandedWG .....	2426
STKBeeThree .....	2428
STKBlowBotl .....	2430
STKBlowHole .....	2432
STKBowed .....	2434
STKBrass .....	2436
STKClarinet .....	2438
STKDrummer .....	2440
STKFlute .....	2442
STKFMVoices .....	2444
STKHevyMetl .....	2446
STKMandolin .....	2448
STKModalBar .....	2450
STKMoog .....	2452
STKPercFlut .....	2454
STKPlucked .....	2456
STKResonate .....	2458
STKRhodey .....	2460
STKSaxofony .....	2462
STKShakers .....	2464
STKSimple .....	2466
STKSitar .....	2468
STKStifKarp .....	2470
STKTubeBell .....	2472
STKVoicForm .....	2474
STKWhistle .....	2476
STKWurley .....	2478
strchar .....	2480
strchark .....	2482
strcpy .....	2483
strcpyk .....	2484
strcat .....	2486
strcatk .....	2488
strcmp .....	2490
strcmpk .....	2492
streson .....	2493
strfromurl .....	2495
strget .....	2497
strindex .....	2499
strindexk .....	2500
strlen .....	2502
strlenk .....	2503
strlower .....	2504
strlowerk .....	2506
strrindex .....	2507
strrindexk .....	2509
strset .....	2510
strsub .....	2512

strsubk .....	2514
strtod .....	2515
strtodk .....	2516
strtol .....	2517
strtolk .....	2518
strupper .....	2519
strupperk .....	2520
subinstr .....	2521
subinstrinit .....	2524
sum .....	2525
sumarray .....	2527
svfilter .....	2529
syncgrain .....	2532
syncloop .....	2535
syncphasor .....	2537
system .....	2541
tb .....	2543
tab .....	2546
tabifd .....	2548
table .....	2550
table3 .....	2552
tablecopy .....	2553
tablefilter .....	2556
tablefilteri .....	2558
tablegpw .....	2560
tablei .....	2561
tableicopy .....	2564
tableigpw .....	2565
tableikt .....	2566
tableimix .....	2569
tableiw .....	2571
tablekt .....	2575
tablemix .....	2578
tableng .....	2580
tablera .....	2582
tableseg .....	2585
tableshuffle .....	2587
tablew .....	2590
tablewa .....	2593
tablewkt .....	2596
tablexkt .....	2598
tablexseg .....	2601
tabmorph .....	2603
tabmorpha .....	2606
tabmorphak .....	2608
tabmorphi .....	2611
tabplay .....	2614
tabrec .....	2615
tabsum .....	2616
tab2pvs .....	2618
tambourine .....	2619
tan .....	2621
tanh .....	2623
taninv .....	2625

The Canonical Csound  
Reference Manual

---

taninv2 .....	2627
tbvcf .....	2629
tempest .....	2632
tempo .....	2635
temposcal .....	2637
tempoval .....	2639
tigoto .....	2641
timedseq .....	2643
timeinstk .....	2646
timeinsts .....	2648
timek .....	2650
times .....	2652
timeout .....	2655
tival .....	2657
tlineto .....	2659
tone .....	2661
tonek .....	2663
tonex .....	2665
trandom .....	2667
tradsyn .....	2669
transeg .....	2671
transegb .....	2673
transegr .....	2675
trcross .....	2677
trfilter .....	2679
trhighest .....	2681
trigger .....	2683
trigseq .....	2685
trirand .....	2689
trlowest .....	2691
trmix .....	2693
trscale .....	2695
trshift .....	2697
trsplitt .....	2699
turnoff .....	2701
turnoff2 .....	2703
turnon .....	2706
tvconv .....	2707
unirand .....	2709
until .....	2711
unwrap .....	2713
upsamp .....	2716
urandom .....	2718
urd .....	2721
vactrol .....	2724
vadd .....	2726
vadd_i .....	2729
vaddv .....	2731
vaddv_i .....	2734
vaget .....	2736
valpass .....	2739
vaset .....	2742
vbap .....	2744
vbapmove .....	2747

vbapg .....	2750
vbapgmove .....	2753
vbap16 .....	2756
vbap16move .....	2758
vbap4 .....	2760
vbap4move .....	2763
vbap8 .....	2766
vbap8move .....	2768
vbaplsinit .....	2771
vbapz .....	2774
vbapzmove .....	2776
vcella .....	2778
vco .....	2781
vco2 .....	2784
vco2ft .....	2788
vco2ift .....	2790
vco2init .....	2792
vcomb .....	2795
vcopy .....	2798
vcopy_i .....	2801
vdelay .....	2803
vdelay3 .....	2805
vdelayx .....	2807
vdelayxq .....	2809
vdelayxs .....	2811
vdelayxw .....	2813
vdelayxwq .....	2815
vdelayxws .....	2817
vdelayk .....	2819
vdivv .....	2820
vdivv_i .....	2823
vecdelay .....	2825
veloc .....	2826
vexp .....	2828
vexp_i .....	2831
vexpseg .....	2833
vexpv .....	2835
vexpv_i .....	2838
vibes .....	2840
vibr .....	2842
vibrato .....	2844
vincr .....	2846
vlimit .....	2849
vlinseg .....	2850
vlowres .....	2852
vmap .....	2854
vmirror .....	2856
vmult .....	2857
vmult_i .....	2861
vmultv .....	2863
vmultv_i .....	2866
voice .....	2868
vosim .....	2871
vphaseseg .....	2876

vport .....	2878
vpow .....	2879
vpow_i .....	2883
vpowv .....	2886
vpowv_i .....	2889
vpvoc .....	2891
vrandh .....	2894
vrandi .....	2897
vstaudio, vstaudiog .....	2900
vstbankload .....	2902
vstedit .....	2903
vstinit .....	2905
vstinfo .....	2907
vstmidiout .....	2909
vstnote .....	2911
vstparamset,vstparamget .....	2913
vstprogset .....	2915
vsubv .....	2916
vsubv_i .....	2919
vtable1k .....	2921
vtablei .....	2923
vtablek .....	2925
vtablea .....	2927
vtablewi .....	2929
vtablewk .....	2930
vtablewa .....	2932
vtabi .....	2934
vtabk .....	2936
vtaba .....	2938
vtabwi .....	2940
vtabwk .....	2941
vtabwa .....	2942
vwrap .....	2943
waveset .....	2944
websocket .....	2946
weibull .....	2948
wgbow .....	2951
wgbowedbar .....	2953
wgbrass .....	2955
wgclar .....	2957
wgflute .....	2959
wgpluck .....	2961
wgpluck2 .....	2964
wguide1 .....	2966
wguide2 .....	2969
while .....	2972
wiiconnect .....	2974
wiidata .....	2976
wiirange .....	2979
wiisend .....	2980
window .....	2982
wrap .....	2985
writescratch .....	2987
wterrain .....	2989

The Canonical Csound  
Reference Manual

---

xadsr .....	2991
xin .....	2993
xout .....	2995
xscanmap .....	2997
xscansmap .....	3000
xscans .....	3001
xscanu .....	3005
xtratim .....	3009
xyin .....	3013
xyscale .....	3015
zacl .....	3017
zakinit .....	3019
zamod .....	3022
zar .....	3024
zarg .....	3026
zaw .....	3028
zawm .....	3030
zdf_1pole .....	3033
zdf_1pole_mode .....	3035
zdf_2pole .....	3037
zdf_2pole_mode .....	3039
zdf_ladder .....	3041
zfilter2 .....	3043
zir .....	3045
ziw .....	3047
ziwm .....	3049
zkcl .....	3051
zkmod .....	3053
zkr .....	3055
zkw .....	3057
zkwm .....	3059
Score Statements and GEN Routines .....	3062
Score Statements .....	3062
a Statement (or Advance Statement) .....	3063
b Statement .....	3065
C Statement .....	3067
d Statement (De Note Statement) .....	3069
e Statement .....	3071
f Statement (or Function Table Statement) .....	3073
i Statement (Instrument or Note Statement) .....	3075
m Statement (Mark Statement) .....	3079
n Statement .....	3081
q Statement .....	3083
r Statement (Repeat Statement) .....	3085
s Statement .....	3087
t Statement (Tempo Statement) .....	3089
v Statement .....	3091
x Statement .....	3093
y Statement (or Seed Statement) .....	3095
{ Statement .....	3097
} Statement .....	3100
GEN Routines .....	3100
GEN01 .....	3104
GEN02 .....	3107

The Canonical Csound  
Reference Manual

---

GEN03 .....	3109
GEN04 .....	3112
GEN05 .....	3115
GEN06 .....	3117
GEN07 .....	3119
GEN08 .....	3121
GEN09 .....	3123
GEN10 .....	3126
GEN11 .....	3128
GEN12 .....	3130
GEN13 .....	3133
GEN14 .....	3137
GEN15 .....	3140
GEN16 .....	3148
GEN17 .....	3151
GEN18 .....	3153
GEN19 .....	3156
GEN20 .....	3158
GEN21 .....	3161
GEN23 .....	3165
GEN24 .....	3167
GEN25 .....	3169
GEN27 .....	3171
GEN28 .....	3173
GEN30 .....	3176
GEN31 .....	3178
GEN32 .....	3179
GEN33 .....	3181
GEN34 .....	3184
GEN40 .....	3187
GEN41 .....	3189
GEN42 .....	3191
GEN43 .....	3193
GEN49 .....	3194
GEN51 .....	3196
GEN52 .....	3199
GENtanh .....	3202
GENexp .....	3204
GENsone .....	3206
GENquadbezier .....	3208
GENfarey .....	3211
GENwave .....	3216
GENpadsynth .....	3219
Experimental Orchestra Opcodes and GEN Routines .....	3223
Experimental Orchestra Opcodes .....	3223
cudanal .....	3224
cudasynth .....	3227
cudasliding .....	3229
Deprecated Orchestra Opcodes and GEN Routines .....	3231
Deprecated Orchestra Opcodes .....	3231
abetarand .....	3232
abexprnd .....	3233
acauchy .....	3234
aexprand .....	3235

The Canonical Csound  
Reference Manual

---

agauss .....	3236
agogobel .....	3237
alinrand .....	3238
apcauchy .....	3239
apoisson .....	3240
apow .....	3241
array .....	3242
atrirand .....	3244
aunirand .....	3245
aweibull .....	3246
bformdec .....	3247
bformenc .....	3249
clock .....	3251
hrtfer .....	3252
ibetarand .....	3254
ibexprnd .....	3255
icauchy .....	3256
ictrl14 .....	3257
ictrl21 .....	3258
ictrl7 .....	3259
iexprand .....	3260
igauss .....	3261
ilinrand .....	3262
imidic14 .....	3263
imidic21 .....	3264
imidic7 .....	3265
instimek .....	3266
instimes .....	3267
ioff .....	3268
ion .....	3269
iondur2 .....	3270
iondur .....	3271
ioutat .....	3272
ioutc14 .....	3273
ioutc .....	3274
ioutpat .....	3275
ioutpb .....	3276
ioutpc .....	3277
ipcauchy .....	3278
ipoisson .....	3279
ipow .....	3280
is16b14 .....	3281
is32b14 .....	3282
islider16 .....	3283
islider32 .....	3284
islider64 .....	3285
islider8 .....	3286
itablecopy .....	3287
itablegpw .....	3288
itablemix .....	3289
itablew .....	3290
itrirand .....	3291
iunirand .....	3292
iweibull .....	3293

kbetarand .....	3294
kbexprnd .....	3295
kcauchy .....	3296
kdump2 .....	3297
kdump3 .....	3298
kdump4 .....	3299
kdump .....	3300
kexprand .....	3301
kfilter2 .....	3302
kgauss .....	3303
klinrand .....	3304
kon .....	3305
koutat .....	3306
koutc14 .....	3307
koutc .....	3308
koutpat .....	3309
koutpb .....	3310
koutpc .....	3311
kpcauchy .....	3312
kpoisson .....	3313
kpow .....	3314
kread2 .....	3315
kread3 .....	3316
kread4 .....	3317
kread .....	3318
ktableseg .....	3319
ktrirand .....	3320
kunirand .....	3321
kweibull .....	3322
sndload .....	3323
peakk .....	3325
pop .....	3326
pop_f .....	3328
push .....	3329
push_f .....	3331
soundout .....	3332
soundouts .....	3334
stack .....	3335
Deprecated Gen Routines .....	3336
GEN22 .....	3337
The Utility Programs .....	3338
Directories .....	3338
Soundfile Formats .....	3338
Analysis File Generation (ATSA, CVANAL, HETRO, LPANAL, PVANAL) ...	3339
File Queries (SNDINFO) .....	3349
File Conversion (HET_IMPORT, HET_EXPORT, PVLOOK, PV_EXPORT, PV_IMPORT, SDIF2AD, SRCONV) .....	3350
Other Csound Utilities (CS, CSB64ENC, ENVEXT, EXTRACTOR, MAKECSD, MIXER, SCALE, MKDB) .....	3367
Cscore .....	3381
Events, Lists, and Operations .....	3381
Writing a Cscore Control Program .....	3384
Compiling a Cscore Program .....	3388
More Advanced Examples .....	3391

Csbeats .....	3393
IV. Opcode Quick Reference .....	3398
Opcode Quick Reference .....	3400
A. List of examples .....	3460
B. Pitch Conversion .....	3503
C. Sound Intensity Values .....	3507
D. Formant Values .....	3508
E. Modal Frequency Ratios .....	3512
F. Window Functions .....	3514
G. SoundFont2 File Format .....	3519
H. Csound Double (64-bit) vs. Float (32-bit) .....	3520
Glossary .....	3521

---

# Preface

## Table of Contents

Preface to the Csound Manual .....	xxxv
History of the Canonical Csound Reference Manual .....	xxxvi
Copyright Notice .....	xxxvii
Getting Started with Csound .....	xxxix
What's new in Csound 6.09.1 .....	xli

## Preface to the Csound Manual

Barry Vercoe, MIT Media Lab

Realizing music by digital computer involves synthesizing audio signals with discrete points or samples representative of continuous waveforms. There are many ways to do this, each affording a different manner of control. Direct synthesis generates waveforms by sampling a stored function representing a single cycle; additive synthesis generates the many partials of a complex tone, each with its own loudness envelope; subtractive synthesis begins with a complex tone and filters it. Non-linear synthesis uses frequency modulation and waveshaping to give simple signals complex characteristics, while sampling and storage of a natural sound allows it to be used at will.

Since comprehensive moment-by-moment specification of sound can be tedious, control is gained in two ways: 1) from the instruments in an orchestra, and 2) from the events within a score. An orchestra is really a computer program that can produce sound, while a score is a body of data which that program can react to. Whether a rise-time characteristic is a fixed constant in an instrument, or a variable of each note in the score, depends on how the user wants to control it.

The instruments in a Csound orchestra (see *Syntax of the Orchestra*) are defined in a simple syntax that invokes complex audio processing routines. A score (see *The Standard Numeric Score*) passed to this orchestra contains numerically coded pitch and control information, in standard numeric score format. Although many users are content with this format, higher level score processing languages are often convenient.

The programs making up the Csound system have a long history of development, beginning with the Music 4 program written at Bell Telephone Laboratories in the early 1960's by Max Mathews. That initiated the stored table concept and much of the terminology that has since enabled computer music researchers to communicate. Valuable additions were made at Princeton by the late Godfrey Winham in Music 4B; my own Music 360 (1968) was very indebted to his work. With Music 11 (1973) I took a different tack: the two distinct networks of control and audio signal processing stemmed from my intensive involvement in the preceding years in hardware synthesizer concepts and design. This division has been retained in Csound.

Because it is written entirely in C, Csound is easily installed on any machine running Unix or C. At MIT it runs on VAX/DECstations under Ultrix 4.2, on SUNs under OS 4.1, SGI's under 5.0, on IBM PC's under DOS 6.2 and Windows 3.1, and on the Apple Macintosh under ThinkC 5.0. With this single language for defining the audio signal processing, and portable audio formats like AIFF and WAV, users can move easily from machine to machine.

The 1991 version added phase vocoder, FOF, and spectral data types. 1992 saw MIDI converter and control units, enabling Csound to be run from MIDI score-files and external keyboards. In 1994 the sound analysis programs (lpc, pvoc) were integrated into the main load module, enabling all Csound processing to be run

from a single executable, and Cscore could pass scores directly to the orchestra for iterative performance. The 1995 release introduced an expanded MIDI set with MIDI-based linseg, butterworth filters, granular synthesis, and an improved spectral-based pitch tracker. Of special importance was the addition of run-time event generating tools (Cscore and MIDI) allowing run-time sensing and response setups that enable interactive composition and experiment. It appeared that real-time software synthesis was now showing some real promise.

# History of the Canonical Csound Reference Manual

This initial version of this manual for early versions of Csound was started at MIT by Barry L. Vercoe and maintained there during the 1980's and start of the 1990's. Some of the manual comes from documents for programs like *Music11* from the 1970's. This original manual was improved and worked on by Richard Boulanger, John ffitich, Jean Piché and Rasmus Ekman.

This manual led to the Official Csound Reference Manual, still located at: <http://www.lakewood-sound.com/csound> [<http://www.lakewoodsound.com/csound/hypertext/manual.htm>], for Csound version 4.16, November, 1999, which was maintained by David M. Boothe.

A parallel version of the manual called the Alternative Csound Reference Manual, was developed by Kevin Conder using *DocBook/SGML* [<http://www.docbook.org/>]. This version later became the Canonical version.

When Csound was licenced as LGPL by MIT in 2003, the manual was licenced GFDL and placed on Sourceforge along with the sources of Csound.

In the winter of 2004, the Canonical Manual was converted to DocBook/XML by Steven Yi to allow for more people to be able to compile and maintain the manual.

The manual is currently maintained by Andrés Cabrera with continuous contributions from the Csound Community.

The manual continues to be a community run project that depends on the contributions of developers and users to help refine the coverage and accuracy of its contents. All contributions are welcome and appreciated.

## Table 1. Other Contributors

Mike Berry
Eli Breder
Michael Casey
Michael Clark
Perry Cook
Sean Costello
Richard Dobson
Mark Dolson
Dan Ellis
Tom Erbe
Bill Gardner

Michael Gogins

Matt Ingalls

Richard Karpen

Anthony Kozar

Victor Lazzarini

Allan Lee

David Macintyre

Gabriel Maldonado

Max Mathews

Hans Mikelson

Peter Neubäcker

Peter Nix

Ville Pulkki

Maurizio Umberto Puxeddu

John Ramsdell

Marc Resibois

Rob Shaw

Paris Smaragdis

Greg Sullivan

Istvan Varga

Bill Verplank

Robin Whittle

Steven Yi

François Pinot

Andrés Cabrera

Gareth Edwards

Joachim Heintz

John ffitch

Oeyvind Brandtsegg

Menno Knevel

Felipe Sateler

And many others.

This list is by no means complete. More information can be gathered from the Changelog file in the manual's sources repository.

## Copyright Notice

This version of the Csound Manual ("The Canonical Csound Manual") is released under the GNU Free Documentation Licence [<http://www.gnu.org/licenses/fdl.txt>]. Below are listed, for historical purposes, previous copyrights and requests for credit from previous authors.

## Previous copyright notices

Copyright (c) 1986, 1992 by the Massachusetts Institute of Technology. All rights reserved.

Developed by *Barry L. Vercoe* at the Experimental Music Studio, Media Laboratory, M.I.T., Cambridge, Massachusetts, with partial support from the System Development Foundation and from National Science Foundation Grant # IRI-8704665.

## Manual

Copyright (c) 2003 by Kevin Conder for modifications made to the Public Csound Reference Manual.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of this license is available in the examples sub-directory [examples/fdl.txt] or at: [www.gnu.org/licenses/fdl.txt](http://www.gnu.org/licenses/fdl.txt) [<http://www.gnu.org/licenses/fdl.txt>].

This Csound language documentation in this manual is derived from Kevin Conder's *Alternative Csound Reference Manual*, which in turn is derived from the *Public Csound Reference Manual*.

Copyright 2004-2005 by Michael Gogins for modifications made to the *Alternative Csound Reference Manual*.

This legal notice is from the *Public Csound Reference Manual*: "The original Hypertext Edition of the MIT Csound Manual was prepared for the World Wide Web by *Peter J. Nix* of the Department of Music at the University of Leeds and *Jean Piché* of the Faculté de musique de l'Université de Montréal. A Print Edition, in Adobe Acrobat format, was then maintained by *David M. Boothe*. The editors fully acknowledge the rights of the authors of the original documentation and programs, as set out above, and further request that this notice appear wherever this material is held."

The Public Csound Reference Manual's last known network location was <http://www.lakewood-sound.com/csound/hypertext/manual.htm>.

The Alternative Csound Reference Manual's network location, for both the Transparent and Opaque copies, is <http://kevindumpscore.com/download.html#csound-manual>.

The Csound and CsoundAC Manual's network location is <http://sourceforge.net/projects/csound>.

## Csound and CsoundAC

Csound is copyright 1991-2008 by Barry Vercoe, John ffitich and others.

CsoundAC is copyright 2001-2008 by Michael Gogins.

Csound and CsoundAC (formerly CsoundVST) are free software; you can redistribute them and/or modify them under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

Csound and CsoundAC are distributed in the hope that they will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with Csound and CsoundAC; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

---

# Virtual Synthesis Technology

Virtual Synthesis Technology (VST) PlugIn interface technology by Steinberg Soft- und Hardware GmbH.

## Getting Started with Csound

### Downloading

In case you don't already have Csound (or have an older version) download the appropriate Csound version for your platform from the *Sourceforge Csound Download Page* [<http://sourceforge.net/projects/ksound/files/>]. Installers for Windows have '.exe' extension and for Mac '.dmg'. If the installer's filename ends in '-d' it means the installer has been built with *double* precision (64-bit) which provides higher quality output than the ordinary *float* precision (32-bit). The float versions provide quicker output, which may be important if you're using Csound in a real-time setting. You can also download the sources and build them, but this requires more expertise (See the section *Building Csound*).

It may also be useful to download the most recent version of this manual, which you will also find there.

### Running

Csound can be run in different ways. Since Csound is a command line program (DOS in Windows terms), just clicking on the csound executable will have no effect. Csound must be called either from the computer's command line or from a front end. To use Csound from the command line, you must open a *Terminal* (Command Prompt or DOS Prompt on Windows, or Terminal on MacOS). Using Csound from the command line can be difficult if you've never used a terminal, so you may want to try to use one of the front ends, either QuteCsound, which is included with the latest distributions, or another front end. A *front end* is a window-based (not necessarily Windows-based) program that assists running Csound. Most front ends include text editors with which you can edit csound files, and many include other useful features.

Whether being run from a front end or being executed from the command line, Csound needs two things:

- A Csound file ('.csd' or possibly an '.orc' and a '.sco' file)
- A list of command line flags (or configuration options) that configure execution. They determine things like output filename and format, whether real-time audio and MIDI are enabled, which audio output to use for real-time audio, the buffer size, the types of messages printed, etc. These options can be included in the '.csd' file itself, so for the examples included in this manual *you shouldn't need to worry about them*. Front end programs often have dialog boxes in which the command line flags can be set. The complete and very long list of available command flags can be found *here*, but you might want to have a look there later...

See the section *Configuring* if Csound is giving you trouble.

This documentation includes many '.csd' files which you can try out, and which should work directly from the command line or from any front end. A simple example is *oscil.csd* [examples/oscil.csd], which can be found in the *examples* folder of this documentation. Your front end should allow you to load the file, and the front end should have a 'play' or 'render' button that will allow you to hear the file. If you want to experiment with the file, you're well advised to use the front end's 'Save As...' command to copy it to some other directory on your hard drive, such as a 'csound scores' directory that you create.



#### Note for MacCsound users

You might need to remove all the lines from the command options slot in order for the manual examples to work.

You can also try the manual examples from the command line. To do this, navigate to the examples directory of the manual using something like this on Windows (assuming the manual is located at `c:\Program Files\Csound\manual\`):

```
cd "c:\Program Files\Csound\doc\manual\examples"
```

or something like:

```
cd /manualdirectory/manual/examples
```

for the Mac or linux Terminal. Then type:

```
csound oscil.csd
```

The example files are configured to run in real time by default, so with this command you should hear a two-second sine wave.

## Writing your own .csd files

A `.csd` file looks like this (this file is `oscils.csd` [examples/oscils.csd]):

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o oscils.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

iflg = p4
asig oscils .7, 220, 0, iflg
outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 2 0
i 1 3 2 2 ;double precision
e
</CsScore>
</CsoundSynthesizer>
```

Csound's `.csd` files have three main sections between the `<CsoundSynthesizer>` and `</CsoundSynthesizer>` tags:

- *CsOptions* - Includes the *Command Line flags* specific to this particular file. These options can also be set using the `.csoundrc` file, which you can edit in a text editor, or directly in the *command line*. Some front ends also provide ways to specify global or local options.
- *CsInstruments* - Contains the instruments or processes available in the file. Instruments are defined using the *instr* and *endin* opcodes. The *CsInstruments* section also contains the *Orchestra Header*, which

defines things like *sample rate*, the *number of samples in a control period*, and the *number of output channels*.

- *CsScore* - Contains the 'notes' to be played, and optionally the definition of f-tables. Notes are created using the *i statement*, and f-tables are created using the *f statement*. Several other *score statements* are available.

Anything after a semicolon (;) until the end of the line is a comment, and is ignored by Csound.

You can write .csd files in any plain text editor, such as Notepad or Textedit. If you use a word processor (not recommended), be sure to save the file as plain text (not rich text). Many *front ends* include advanced editing capabilities, such as syntax highlighting and auto-completion of code.

You can find an in-depth tutorial on getting started with Csound written by Michael Gogins here [<http://michael-gogins.com/archives/tutorial.pdf>].

## What's new in Csound 6.09.1

### Release Notes for Csound 6.09 (2017 May)

A mixed bag of new opcodes and many fixes and improvements.

Also as usual there are a number of internal changes, including many memory leaks fixed and more robust code.

- New opcodes:
  - *select* -- sample-by-sample comparison of audio selecting the output.
  - *midiarp* -- generates arpeggios based on currently held MIDI notes.
  - *hilbert2* -- a DFT-based implementation of a Hilbert transformer.
  - Ableton Link opcodes -- for synchronizing tempo and beat across local area networks.
  - *pvstrace* -- retain only the N loudest bins.
  - several new unary functions/opcodes for k-rate and i-time numeric arrays: *ceil*, *floor*, *round*, *int*, *frac*, *powoftwo*, *abs*, *log2*, *log10*, *log*, *exp*, *sqrt*, *cos*, *sin*, *tan*, *acos*, *asin*, *atan*, *sinh*, *cosh*, *tanh*, *cbirt*, *limit1*.
  - several new binary functions/opcodes for k-rate and i-time numeric arrays: *atan2*, *pow*, *hypot*, *fmod*, *fmax*, *fmin*.
  - *limit* -- numeric limiting within a given range (for arrays).
  - *tvconv* -- a time-varying convolution (FIR filter) opcode.
  - *liveconv* -- partitioned convolution with dynamically reloadable impulse response.
  - *bpf*, *xyscale*, *ntom*, *mton* -- (from SuperCollider?).
  - *OSCsendA* -- asynchronous version of OSCsend.
  - *OSCsend* -- now implemented directly using system sockets. Old version using liblo has been kept as *OSCsend\_lo*.
  - *OSCraw* -- to listen for all OSC messages at a given port.

- *sorta* and *sortd*-- sort elements of an array.
- *dot* -- calculates the dot product of two arrays.
- zero delay filters -- *zdf\_1pole\_mode*, *zdf\_2pole\_mode*, *zdf\_ladder*, *zdf\_1pole* and *zdf\_2pole*, *diode\_ladder*, *K35\_hpf* and *K35\_lpf*.
- *product* -- takes a numeric array (k or i-rate) and calculates its product.
- supercollider ugens -- *sc\_phasor*, *sc\_lag*, *sc\_lagud*, *sc\_trig*.
- Orchestra:
  - Including a directory of UDO files no longer fails if more than about 20 entries.
  - It was possible for kr, sr, and ksmps to be inconsistent in one case, no more.
  - Macro names better policed and bracket matching.
  - Octal values as \000 can be in strings.
  - (from 6.09.1) In a UDO the out\* opcodes now work, where before it was working only sometimes.
- Score:
  - Improved line number reporting in r opcode and case with no macro implemented.
  - m and n opcodes fixed.
  - Expansion of [...] corrected and improved.
  - Strings in scores improved.
  - The ) character can be in a macro argument if it is escaped with \.
  - Use of the characters e or s could lead to errors; now fixed.
  - Macro names better policed, and bracket matching.
  - p2 and p3 are now at higher precision, no longer truncated to 6 decimal places.
  - new opcode d to switch off infinite notes (denote); same as i with negative p1.
  - named instruments can be turned off with i if a - follows the ".
  - (from 6.09.1) if an r-opcode section ended in e-opcode it used to stop early.
- Options:
  - jack midi module now can report available devices under --midi-devices.
  - (from 6.09.1) defining smacros and omacros on command line only happens once.
  - (from 6.09.1) defining smacros from command line now works.
- Modified Opcodes and Gens:
  - ftgentmp improved string arguments.

- hdf5read opcode now reads entire data sets when dataset name string is suffixed with an asterisk.
  - use of non power-of-two lengths now acceptable where before it was inconsistent.
  - ampmidid optionally can be aware of 0dbfs.
  - dust and dust2 at k-rate now conform to the manual (NOTE: this is an incompatible change).
  - In prints the format %% now prints one %.
  - OSClisten can be used with no data outputs.
  - GEN18 corrected to write to requested range.
  - sockrev now can read strings.
  - vbap system can in some cases allow arbitrary number of speakers via arrays (work in progress).
  - Websocket server can only accept one protocol output, so limiting intype to just a single argument.
  - sum opcode will also sum elements of an array.
  - Overloaded pvs2tab and tab2pvs now can create and use split magnitude and phase arrays.
  - Utilities:
    - dnoise fixed.
  - Frontends:
    - Removed HTML5 Csound editor which has quit working.
    - Emscripten: Emscripten Csound (asm.js) now requires sourcing in CsoundObj.js and FileList.js separately from libcsound.js. This is to accommodate using the same JS API with either asm.js or wasm backends.
    - CsoundQT: CsoundQt 0.9.4 is announced: [https://github.com/CsoundQt/CsoundQt/blob/develop/release\\_notes/Release notes 0.9.4.md](https://github.com/CsoundQt/CsoundQt/blob/develop/release_notes/Release%20notes%200.9.4.md) [[https://github.com/CsoundQt/CsoundQt/blob/develop/release\\_notes/Release%20notes%200.9.4.md](https://github.com/CsoundQt/CsoundQt/blob/develop/release_notes/Release%20notes%200.9.4.md)].
    - Windows installer with CsoundQt includes PythonQt.
  - Bugs Fixed:
    - pwd works on OSX.
    - Fencepost error in sensLine fixed.
    - OSCsend corrected for caching of host name.
    - Bug in push/pop opcodes fixed (this opcode is now a plugin and deprecated).
    - Bug in sprintf removed.
    - Bug in soundin removed.
- 
- losci/losci3 fixed in case of long tables.xliii

- inrg was broke for a while.
- Partikkel channelmask panning laws had an indexing error, now fixed.
- jack audio module now allows for independent numbers of in and out channels.
- Bug in string copying fixed.
- Bug in hdf5read where if two hdf5read opcodes were placed in series in an instrument, the argument names of the second opcode instance would be incorrect due to directly changing the last string character of the first when reading an entire dataset.
- Memory leaks fixed in some plugin opcodes.
- System Changes:
  - soundin now uses the diskin2 code.
  - out family of opcodes reworked to reduce interleaving costs and to take proper regard if nchnls value.
  - (from 6.09.1) a crash on Linux i386 removed relating to server mode.
- API:
  - New csound\_threaded.hpp header-only facility, obviating need for csPerfThread.cpp object in some projects.
  - Added GetA4 function.
  - New framework for plugin opcode development in C++ using Csound's allocator.
  - Added StrDup function.
  - Boost dependencies removed from Csound interfaces, CsoundAC, and CsoundVST.
  - (from 6.09.1) Two new API function, csoundSetSpinSample and csoundClearSpin.
- Platform Specific:
  - iOS
    - iPad portrait SplitView fix+animation, info popover resizing, stop button fix in Soundfile Pitch Shifter.
    - Csound-iOS API updates; Examples cleaned up, enhanced/expanded, and reordered. Manual revised, expanded, updated. Updates to API and examples support iOS 10 and Xcode 8.
  - Android
    - Multichannel input and output allowed.
  - Windows
    - csound64.lib import library added to Windows installer.

## Release Notes for Csound 6.08 (2016 November)

As usual there are a number of opcode fixes and improvements, but the major changes are in the language structures. First the score language has all-new treatment of macros and preprocessing, bringing it in line with those of the orchestra. The parsing of the orchestra has had a number of fixes as outlined below.

A major, and not totally compatible change as been made in reading and writing array elements. The rate of the index now often determines the time of processing; check the entry below under Orchestra. This simplifies much code and seems to capture expectations; the earlier ad hoc code had many anomalies.

Also as usual there are a number of new opcodes and internal fixes to memory leaks and more robust code.

- New opcodes:
  - *dct*: Discrete Cosine Transform of a sample array (type-II DCT).
  - *getfargs*: copy arguments of a gen to an S-variable.
  - *mfb*: implements a mel-frequency filterbank for an array of input magnitudes.
- New Gen and Macros:
  - *quadbezier*: generating Bezier curves in a table.
- Orchestra:
  - The character  $\neg$  is now correctly treated as a variant of  $\sim$  for bitwise not.
  - Lexing bug which could corrupt strings fixed.
  - Ensure no newlines in string-lexing.
  - Small improvement in reported line numbers.
  - Better checking of macro syntax.
  - Improved parsing of setting of labels.
  - Added error handling for unmatched brackets for UDO arg specification.
  - Check that #included file is not a directory.
  - Deeply nested macro calls better policed.
  - For years Csound has fixed the pitch of A4 at 440Hz. Now this can be set in the header using the new r-variable A4, and also read with that variable.
  - Floating point values can use e or E for exponent.
  - Array access semantics have been clarified:
    - $i[i] \Rightarrow$  reading at i-time and perf-time, writing at i-time only.
    - $i[k] \Rightarrow$  reading at perf-time, writing yields a runtime error.
    - $k[i], k[k] \Rightarrow$  reading at perf-time, writing at perf-time.
    - $a[i], a[k] \Rightarrow$  reading at perf-time, writing at perf-time.

- other (S[], f[]) => reading and writing according to index type (i,k).  
In particular, i(k[i]) will continue not to work, as before, but the new operator i(k[],i) is provided to cover this case.
- xout validation no longer fails when constants are given.
- Score:
  - New code to handle macros and other preprocessor commands. Brings it into line with orchestra code.
  - New score opcode C introduced as a way of switching automatic carry off (C 0) or on (default) (C 1).
- Options:
  - The tempo setting can now be a floating point value (previously fixed to integer).
  - New option --version prints version information and exits.
- Modified Opcodes and Gens:
  - Problems in centroid fixed.
  - Better treatment of rounding in printks.
  - OSC extended to include multicast.
  - Faust opcodes brought up to date with faust.
  - oscil1 and oscili can take a negative duration.
  - fout opcode documentation clarified.
  - Release time in mxadsr fixed.
  - centroid opcode extended to take array inputs in addition.
  - ptable opcodes are now identical to table family.
  - ftgen now as array input option.
  - subinstr can now have string arguments.
  - the i() format is extended to work on k-rate arrays with the first argument being an array, followed by the indices.
- Utilities:
  - pvlook now always prints explicit analysis window name.
- Frontends:
  - HTML5
    - csound.node: Implemented for Linux, minor API fix.
    - pnacl: Added compileCsdText method to csound object.
- General Usage:

- Checking of valid macro names improved.
- #undef fixed.
- Bugs Fixed:
  - Fixes to prints in format use.
  - jitter2 reworked to make it more like the manual.
  - oscbank has had multiple fixes and now works as advertised.
  - bformdec1 with arrays and type 4 fixed.
  - Bug in pvsceps fixed.
  - In various formatted print opcodes extra trash characters might appear -- fixed.
  - Assigning variables with --sample-accurate could give unexpected results; this is believed fixed now.
  - padsynth square profile fix, and opcode prints less depending on warn level.
  - gen31 fixed.
  - gen41 fixed.
  - Bug in sensekey fixed.
  - A number of issues in centroid fixed.
- System changes:
  - New score lexing and preprocessor.
  - MAC line endings now work again.
  - System information messages (system sampling rate, etc) are now directed to stdout.
  - rtjack reworked to deal with names and numbers.
  - The version printing now includes the commit as so the developers know which patches have been applied.
- API:
  - API version now 4.0.
  - Now supports named gens.
  - fterror now in API.
  - API functions SetOutput and GetOutputFormat fixed.
  - Many API functions now use const where appropriate.
  - Messages can now be directed to stdout from the API by using `CSOUNDMSG_STDOUT` attribute.

- New Lisp CFFI and FFI interfaces tested with Steel Bank Common Lisp (64 bit CPU architecture), runs in separate thread.
- `ctcsound.py`, a new FFI interface for Python was introduced in version 6.07. It is now the recommended interface for Python, `csnd6.py` being deprecated.
- Platform Specific:
  - Android.
    - Multichannel input and output allowed.
  - Windows.
    - `csound64.lib` import library added to Windows installer.

## Release Notes for Csound 6.07 (2016 March)

A large number of bug fixes, some quite major, some internal are included, as well as some new facilities and extensions. As ever there are coding improvements as well.

- New opcodes:
  - *compress2*: like *compress* but using a more normal use of dB (0.0 for full scale).
  - (Experimental, source code-only) New cuda opcodes: *cuda synth2* and *cuda anal2*.
  - *directory* opcode: reads a directory and outputs to a string array a list of file names.
  - *ftsamplerbank*: to load a sample library from a directory.
  - *mvclpf1*, *mvclpf2*, *mvclpf3*, *mvclpf4*, *mvchpf*: Moog voltage-controlled filter emulations from Fons Andriaensen.
  - *S()* converter from k-rate and i-time number to a string.
  - *cepsinv* opcode to calculate the inverse cepstrum of an array.
  - *moogladder2*, is a faster, less accurate implementation of *moogladder*.
  - *paulstretch* opcode is a lightweight implementation of the PaulStretch time-stretching algorithm by Nasca Octavian Paul. It is ideal for timestretching a signal by very large amounts.
  - *mp3scal* implements phase-locked vocoder processing from mp3-format disk files, resampling if necessary.
  - *filesca* implements phase-locked vocoder processing from disk files, resampling if necessary.
- Orchestra:
  - The boolean `? .. : ..` construction can now have string results, while previously it only allowed numbers.
  - The line number reported when an error is detected at the end of a line is now correct. Also more improvements in line numbers for complex syntax.
  - *while* loop improved/fixed.

- Better and consistent reading of comments.
- Continuation lines handled better, especially respecting line numbers.
- opcode: *S(.)* syntax now allowed.
- Score:
  - The list of tempo points in score opcode *t* is now arbitrarily long.
  - A stupidity in *r* and *l* opcodes fixed.
- Options:
  - The *-z* option now suppresses deprecated opcodes, unless given a 2 or 3 argument.
  - The new option *--fftlib* controls which real fft library to use internally (FFTLIB = 0, PFFFT = 1, vDSP = 2)
- Modified Opcodes and Gens:
  - In *OSC* opcodes it is now possible to send and receive arrays, tables and audio.
  - Better diagnostic if *diskin2* fails.
  - *rezzy* now checks for unstable filter and modifies to close stable version.
  - *adsr* rewritten so it gives an error if the segments are longer than p3.
  - Use of *diskin* to an array now resizes the output array if necessary.
  - *chnget* now checks for a change in channel name as well as data.
  - *interp* can take an optional extra argument to give an initial value.
  - *oscilikts* uses a more liberal table lookup.
  - opcode *in* can read mono or stereo, obviating the *in/ins* distinction.
  - *sensekey* rewritten to provide better diagnostics and fixes a minor bug.
  - Fix to a case of defining a macro with arguments.
  - *sockrecv* now works at a and k rate.
  - *GEN49* now works from *ftgen* calls.
  - *GEN34* liberalised in tables it accepts.
  - *chnget* now allows channel names to be changed at perf time.
  - *iceps* has been renamed *cepsinv* to avoid name clashes.
  - mp3 support improved in a number of minor ways.
  - A minor fix to allow a *ladspa* plugin to be re-loaded by the host.
- Frontends:

- csdebugger:
  - Some memory issues fixed.
- HTML5:
  - CsoundQT has its own notes at [https://github.com/CsoundQt/CsoundQt/blob/develop/release\\_notes/Release%20Notes%200.9.2.1.md](https://github.com/CsoundQt/CsoundQt/blob/develop/release_notes/Release%20Notes%200.9.2.1.md) [[https://github.com/CsoundQt/CsoundQt/blob/develop/release\\_notes/Release%20Notes%200.9.2.1.md](https://github.com/CsoundQt/CsoundQt/blob/develop/release_notes/Release%20Notes%200.9.2.1.md)].
- General usage:
  - The multicore options somewhat improved.
  - When replacing instruments the new version inherits *maxalloc* and active flags.
  - Multicore code now works with midi instruments.
  - MIDI operations now available via a new rtmidi jack module (-+rtmidi=jack).
- Bugs fixed:
  - Fix to *trigseq*.
  - Major error in *rezzy* fixed.
  - *p()* fixed for high numbers of p-arguments.
  - *p()* now works from MIDI events.
  - The 31-bit random number generator could give a very short loop if seeded with zero; fixed.
  - Macros in .orc files now work.
  - A long-standing bug in display of graphs fixed.
  - Fixes to the envelope opcodes *linen*, *expsegr*, *linsegr*, *cossegr*, *transegr*, *envlpx*, including new warnings.
  - Many fixes in string opcodes (*strsub*, *strcpy* and others).
  - Fixed bugs in print formats (*sprintf*, *printf*, *prints*).
  - *pvs*smooth had a missing initialisation.
  - Array initialisation now robust.
  - Bug in *copya2ftab* fixed.
  - Fix in *cusdapsanal* (EXPERIMENTAL).
  - *partikkel* fix to subsample grain placement.
  - Opcodes that require an odd number of arguments are now properly checked.
  - *pvs*write now takes proper notice of 0dbfs.
  - *GEN34* now allowing non-power-of-two source tables.

- *vst4cs* will now work on 64bit architecture.
- *strcat* mended.
- *nstance* mended.
- fixed small issues in *rtauhal* module.
- Windows installer for 64 bit CPU architecture now sets environment variables on system level as it should.
- transeg at k-rate with *itype* non-zero had an initial delay if one cycle.
- log function on arrays fixed.
- Rescaling of named GENs fixed.
- A fencepost error in reading ATS files believed fixed; it generated spurious errors about file finishing early.
- Initialisation error in *pvbandp* fixed.
- *readfi* string allocation fixed.
- System changes:
  - Extensive use made of in-memory files. *Orc* and *sco* are always thus, as are *.csound6rc*. This simplifies much internally.
  - Nested *longjumps* fixed which affected some API use.
  - If a non-existent *.csd* file is presented Csound no longer crashes.
  - The *kcounter* value is now consistently an unsigned 64bit value, allowing for long performances.
- API:
  - *csoundCompileCsdText*: New function to compile a CSD from a string of text.
  - The function call *csound->GetKcounter(csound)* now returns an unsigned 64 bit integer. Previously it returned a long which is unclear.
  - *ctcsound.py* is a new wrapper file to the Csound API for the Python language. It is a pure Python file using the FFI (Foreign Function Interface) module *ctypes*. It does not depend on *Swig*, and it works with Python2 and Python3 as well. It uses *numpy* for its data structures, *numpy* being the fundamental package for scientific computing with Python.
- Platform Specific:
  - Windows.
    - The Windows installer for Csound now contains all executables built for 64 bit CPU architecture.
    - The Windows installer for 64 bit CPU architecture now provides *NW.js* "out of the box." It runs all features of Csound, as well as all features of *HTML5*, in a browser-like runtime with JavaScript scripting, and includes a Csound editor implemented in *HTML*. The editor will run pieces ei-

ther as .csd files with embedded <html> or as .html files calling the csound object, and includes JavaScript debugging.

- A new simplified build system has been implemented in the csound/mingw64 directory for building Csound using the MSYS2/mingw64 toolchain.
- Linking to pthreads and other libraries now static.
- Some limited support for non-ASCII characters in file names.
- OSX.
  - Installer fixed for link names for \_csnd6 and \_CsoundAC.
- GNU/Linux.
  - *date* opcode more accurate.

## Release Notes for Csound 6.06 (2015 September)

A number of bug fixes, some quite major, are included, as well as some new facilities and extensions.

- New opcodes:
  - *getseed* reads the state of the PRN generator; opposite of *seed* opcode.
  - *tabifd* — Instantaneous Frequency Distribution, magnitude and phase analysis.
  - *websocket* — Read and write signals and arrays using a websocket connection.
  - *framebuffer* — Read audio signals into 1 dimensional k-rate arrays and vice-versa with a specified buffer size.
  - *olabuffer* — Sum overlapping frames of audio as k-rate arrays and read as an audio signal
- Orchestra:
  - Labels are allowed in instrument 0.
- Score:
  - Maximum string length in scores increased to 1024.
- Modified Opcodes and Gens:
  - *diskin2* array version uses array size to give number of channels in raw file form.
  - *diskin2* now has the *kpitch* parameter defaulting to 1 for simple use.
  - Vibrato *f-table* in *wgflute* and *wgcar* optional, defaulting to sine wave.
  - *schedule* now accept string arguments.
  - *urandom* now available on OSX platforms.
  - GEN18 had fencepost problem; largely rewritten.
  - In *poscil* family of opcodes it is possible to skip initialisation of phase.

- svfilter now can skip initialisation.
- When opening an input file nchncls\_i is used rather than nchnls. This is a change that should have followed the existence of nchnls\_i
- rtjack module now reports the sampling rate.
- The opcodes rfft, rifft, fft, fftinv, r2c and c2r now have i-rate array versions.
- New optional threshold parameter in opcodes tradsyn, sinsyn and resyn.
- New thresholding option for partials.
- Utilities:
  - extract fixed.
  - src\_conv improved and integrated with -U options.
  - fixes in atsa, and hetl.
- Frontends:
  - pnacl:
    - Support for 48000 sample rate fixed.
  - csound~:
    - Changes to the threading system. String channel initialisation fixed. A number of other bugs were fixed.
  - Emscripten:
    - Csound Javascript object can now receive data from the outvalue opcode.
  - HTML5:
    - Integrate HTML, JavaScript, and other features of HTML5 with Csound, either by embedding a Web page as an <html> element in the CSD file for CsoundQt or Csound for Android, or by hosting Csound in the JavaScript context of a standalone Web browser (Emscripten, PNaCl) or embedded Web browser (csound.node).
- General usage:
  - Ill-formatted macros in the orchestra now trapped.
- Bugs fixed:
  - Use of Windows-style environments for INCDIR etc now works with device numbers.
  - vibrato opcode fixed.
  - Clicking in real-time sample accurate case fixed.
  - Copying of strings now correct; did confuse memory sometimes.
  - Bug in pvstanal fixed.

- Rounding error in `cspch` fixed.
- Removed crash on recompiling a named instrument.
- Fix interpolation bug in `tablexkt`.
- Fix to `plltrack` when `ksmps` is 1.
- System changes:
  - The "error" message from STK plugin is now a warning.
- API:
  - Redefinition of opcodes and UDOs fixed.
- Platform Specific:
  - OSX.
    - `csnd6.jar` link installed in the correct location.
    - Java JNI linking issues solved.
    - fixed link name for `libpng` in `libfltk_image`.

## Release Notes for Csound 6.05 (2015 April)

As ever there are new facilities and numerous bug-fixes. A major part of this release is the removal of a number of memory leaks and over use of memory. Naturally these changes are all but invisible, just a smaller memory foot-print. Note that we track bugs and requests for enhancements via the github issues system, and these had a significant affect on this release.

- Opcodes :
  - The opcode `sndload` is now deprecated.
- New Gen and Macros:
  - Paul Octavian Nasca's `padsynth` algorithm implemented as a gen.
- Score:
  - Fixed string location calculation bug when processing score lines [fixes #443]
- Options:
  - A short-format copyright option is available, with a fixed number of well-known licences (CC, etc)
  - New command-line option to report MIDI devices in simple format
  - New command-line option to set `ksmps`
- Modified Opcodes and Gens:
  - *adsynt* handles amplitude changes better
  - *sfont* has better checking for corruptions

- better checking in physical models for out-of-range frequencies
- *ftgenonce* and others allows string parameters
- *gausstrig* reworked and extended with new features
- use of *p()* function no longer complains overrides the *pcnt* warning
- fix to *midirecv*
- *OSCsend* cleans up after use improved
- *fillarray* is limited to 1 or 2 dimensional arrays; in fact it failed silently previously for 3D and higher.
- *oscbnk* now works when the equaliser is used.
- *mp3in* now works with both mono and stereo input files
- *flooper* & *flooper2* now allow stereo tables
- Release phase of *expsegr* fixed
- f-tables created by a large number of arguments could overwrite memory, now fixed
- performance of *plltrack* improved
- init of arrays clarified and checked
- *gen23* corrected to stop an infinite loop
- *alwayson* now starts from score offset; this is part of a fix to the long-standing problem with *alwayson* in *CsoundVST*
- *invalue* now checks for output string size and reallocates memory if smaller than default string size (set at 256 bytes for backwards compatibility)
- Utilities:
  - The *srconv* utility has been improved but it does not work well, with groups of noise in otherwise good output. We recommend the use of Erik de Castro Lopo's Secret Rabbit Code (aka *libsamplerate*) as providing sample rate conversion at high quality. *srconv* will be removed shortly possibly to be replaced by an SRC-based utility.
- Frontends:
  - *pnac1*
    - Added interface to allow the use of *Csound*'s MIDI input system.
    - Fixed audio input to conform to the latest Pepper API spec.
- Bugs fixed:
  - bugs in *fastabi*, *oscktp*, *phasorbnk*, *adsr*, *xadsr*, *hrtfer* fixed.
  - bugs in the *harmon*. *harmon2*, *harmon3* and *harmon4* fixed.
  - *Csound* could crash after a parsing error, a case now removed.

- System changes:
  - There are now checks that xin/xout types match those defined as part of UDO definition.
  - jack now has a timeout.
- Internal Changes:
  - Many defects indicated by coverity fixed or code changed. Should make csound more robust in edge cases.
  - Parser-related changes simplifies allocation of temporary variables, with some new optimisations.
  - code for multi-thread rendering improved and stabilised vis-a-vis redefinition of instruments.
- Platform Specific:
  - iOS.
    - Fixed audio callback to work correctly with lightning output and Apple TV.
  - Android.
    - New experimental audio IO mode: `csoundPerformKsmmps()` is called from the OpenSL ES output callback. This mode can be optionally enabled by passing a value of "false" to a new second parameter to the `CsoundObj` constructor (bool `isAsync`). The default constructor and the one-parameter sets this to "true" (keeping backwards compatibility with existing code).
    - The OSC opcodes are included in distribution.
    - Android app
      - There are new file open and save dialogs that permit the user to access the SD card on the device, if there is one, in addition to internal storage.
      - There is a new "Save as..." button that permits the user to save the csd as a new file with a new name.
      - Many of the examples in the archive of Android examples are now built into the app and can be run from the app's menu.
      - Includes now the exciter opcode.
  - OSX.
    - Installation now places `csladspa.so` rather than `csladspa.dylib` on disk.
  - Linux.
    - Linux is now build without FLTK threads. This removes system hangs and is in line with other builds.

## Release Notes for Csound 6.04 (2014 November)

---

This new version has many extensions and fixes; many new opcodes and significant numbers of internal reworking. There is a new frontend and iOS ~~and~~ Android version have seen many improvements.

As ever we track bugs and requests for enhancements via the github issues system. Already proposals for the next release are being made but the volume of changes require a release now.

- New opcodes:
    - *pinkier* generates high quality pink noise.
    - power opcode ^ now works with array arguments.
    - *exciter* opcode, modelled on the calf plugin.
    - *vactrol* opcode simulates an analog envelope follower.
    - family of hdf5 opcodes to handle hdf5 format files.
    - (experimental undocumented) buchla opcode models the lowgate filter of Buchla.
  - New k-rate opcodes acting on arrays:
    - transforms: rfft, rifft, fft, fftinv
    - complex product: complxprod
    - polar - rectangular conversion: rect2pol, pol2rect, mags, phs
    - real - complex: r2c, c2r
    - windowing: window
    - cepstrum: pvscpes, iceps, ceps
    - column / row access: getrow, getcol, setrow, setcol
    - a-rate data - k-array copy: shiftin, shiftout
    - phase unwrapping: unwrap
  - New Gen and Macros:
    - Line numbers corrected in instr statements.
    - New control operation, while, for looping.
    - A long-standing bug with macros which use the same name for an argument has been corrected.
    - Redefinition of an instrument in a single call to compile is flagged as an error.
    - ID3 header skip for mp3 files now properly implemented.
    - Errors induced by not defining the location of STK's raw wave files has been removed.
    - bug fixed where UDO's could not read strings from pfields.
  - Modified Opcodes and Gens:
    - stackops opcodes deprecated.
- 
- lenarray extended to handle multi-dimensional arrays.

- ftgenonce accepts string arguments correctly and multiple string arguments.
  - max and min now have initialisation-time versions.
  - gen23 improved regarding comments and reporting problems.
  - in OSCsend the port is now a k-rate value.
  - socksend now works at k-rate.
  - a number of envelope-generating opcodes are now correct in sample-accurate mode.
  - faust compilation is now lock-protected.
  - mp3 fixed to allow reinit to be used with it.
  - In remote opcode the name of the network can be set via the environment variable CS\_NETWORK. Defaults to en0 (OSX) or eth0.
  - Frontends:
    - icsound: New frontend icsound is now ready for general use. icsound is a python interface for interactive work in the ipython notebook.
    - csdebugger: A number of changes and improvements have been made, like stepping through active instruments, better line number use.
  - General usage:
    - Jack module now does not stop Csound if autoconnect fails.
  - Bugs fixed:
    - atsinnoi fixed.
    - ftsavek fixed.
    - sprintf fixed.
    - gen27 fixed, especially with extended arguments, as well as fixed a number of errors in extended score arguments.
    - Physem opcodes (guiro cabasa, sekere) fixed so second call works.
    - flooper fixed in mode 2.
    - OSCsend multiple fixes.
    - UDO fix for case of local ksmps of 1.
    - More changes/fixes to dssi code.
    - xscanu and scanu fixed.
    - temposcal and mincer fixed.
-

- System changes:
  - In server mode exit is now clean.
  - Fixes to rtalsa module.
  - Pulseaudio rt module fixes.
  - Fix to remove fluidEngine entries for csound instance (prevents crash on moduleDestroy).
  - Opcodes called through function calls that returned arrays did not correctly synthesize args as array types due to not converting the arg specifier to the internal format.
  - fixed crashing issue during note initialization for tied notes due to goto skipping over code.
  - fixed incorrect initialization of pfields when note's pfields length were less than instrument expected (off-by-one).
- Internal Changes:
  - Added Runtime Type Identification for instrument variables; removed use of XINCODE/XOUTCO.
  - fix malloc length in negative number parsing, and improved handling of negative numbers.
  - writing to circularBuffer is now atomic.
  - a number of memory leaks and potential dangerous code have been fixed.
  - type-inference has been extensively reworked, as have a few parsing areas.
- API:
  - Added API function for retrieving GEN parameters used for creating a table.
- Platform Specific:
  - iOS.
    - API Refactored for clearer method names and abstraction names (i.e. CsoundBinding instead of CsoundValueCacheable).
    - Updated to remove deprecated code.
    - A significant amount of reworking has been done on the code.
  - Android.
    - API Refactored for clearer method names and abstraction names (i.e. CsoundBinding instead of CsoundValueCacheable).
    - Changes to enable HTML 5 with JavaScript and it is to be hoped WebGL in the Csound6 Android app.
    - Enabled change of screen orientation in the Csound6 app without forcing a restart of the app.
    - Enabled local storage (useful for saving and restoring widget values, etc.).
- Windows.

- fixed pointer arithmetic that caused crashing on Windows.
- pyexec changed to use python's file opening functions to prevent crash on Windows.
- OSX.
  - CsoundAC now compiles.
- Linux.
  - threadlocks bug fix on linux.

## Release Notes for Csound 6.03 (2014 May)

This new version has a large number of bug fixes (including clearing many tickets on SourceForge and GitHub) as well internal changes to improve performance.

- New opcodes:
  - `prinks2`  
: prints a new value every time a control variable changes using a `printf()` style syntax
  - `mp3sr`, `mp3bitrate`, and `mp3chnls` to get information on mp3 files
  - EXPERIMENTAL: CUDA opcodes for partitioned convolution direct convolution and sliding phase vocoding; OpenCL opcode for additive synthesis
  - `compilecsd`  
to compile instruments from a standard CSD file
- Orchestra:
  - The argument for `i()` is supposed to be a variable not an expression. This is now enforced. (bug #90)
- Score:
  - New score opcode `y` sets the random seed (for `~`) at read time
- Options:
  - There was a bug in `CsOptions`; the last argument was missed being read(issue #296)
  - As command-line options `expression-opt` and `no-expression-opt` do nothing in Csound6 a warning is printed
- Modified Opcodes and Gens:
  - For ogg output it is possible to specify a VBR (variable block rate) quality.
  - `dssi4cs` code has been extensively reworked to avoid potential memory faults.
  - Many array operations now available for i-arrays as well as k-arrays.
  - `fillarray` will work for string arrays
  - Displays of FFT (via `disppfft`) improved with scaling/zooming options

- Signal flow graph opcodes are now working with a-rate array signals.
- In also RT code the sample rate is taken from the device
- Faust opcode system updated to latest faust API
- Utilities:
  - fixed bug in lpanal
- csound~:
  - OSX - fix for running with 32-bit cpu architecture
  - Windows - csound~ now available for Windows
- Emscripten:
  - This is now generally merged into the code-base
- General usage:
  - --displays now switches graphs on, as expected
  - New commandline option --get-system-sr added to obtain the machine's sample rate
  - New command-line option --devices[=in|out] gives a list of available audio devices and then exit
- Bug fixes:
  - fixed the bug when tables were replaced but the size did not change
  - A number of bugs in --sample-accurate have been detected and fixed. This includes opcodes *out*, *outn*, and *line*
  - A number of bugs in *grain3* were fixed
  - Bug in *str\_chanel* could cause a crash; fixed
  - Small bug in *rtjack* fixed
  - Error in *resize* opcode corrected
  - Fixed an unlikely bug in *atsa*
  - Fixed *rtauhal* pause issue
  - A number of bugs/untidiness fixed in GEN23
  - Array bound checks fixed
  - strings channels were not correctly set for dynamic-size strings, now fixed
  - memory allocation for string formatting in *printfsk* was fixed, stopping string truncation
  - *strcat* safe against overflow
  - error in compilation of arrays fixed (issue #293)

- *GetPvsChannel* fixed against a crash
- System Changes:
  - *turnoff* opcode now checks that the instrument being affected is active
  - *lenarray* can accept any array type
  - the way of rounding a table number to an integer was changed and is now more as expected
  - there is a new possible section in a csd file called <CsFile...> which is like csFileB but with unencoded text.
  - UDO compilation now uses the type system. This means that UDOs now allow any array type to be used
  - Improved orchestra parsing speeds with better algorithms
- Internal Changes:
  - The whole system has been checked by the Coverity static checker which identified a number of (mainly minor) problems. These have been reviewed and checked. In particular better use of printing and string copying should prevent overflows
  - The type and variable system has been extensively rewritten; this allows better array and UDO support
  - Alignment of variables got right in all cases
  - Array copying is now using the type system to copy values; fixes issues with copying string arrays, f-sigs, etc
  - Always reset Csound when stopping to ensure state is clean; was not being reset when there was a compile error, so that next successful run would start with an invalid Csound engine (issue #305)
- API:
  - All opcodes etc now use the API memory allocation operations, so it is possible to replace the whole memory allocator
  - Added *csoundCompileCsd* to API and associated new *compilecsd* opcode
  - Protected *csoundGetStringChannel* against null and short strings and added a check for string size
  - A number of API functions have had char\* changed to const char\* which reflect the usage

The performance engine now includes debugging capabilities to allow interrupting rendering and providing introspection into the engine's state and instrument variables. The following new functions are available by including the csdebug.h header:

```
void csoundDebuggerInit (CSOUND *csound);
void csoundDebuggerClean (CSOUND *csound);
void csoundSetInstrumentBreakpoint (CSOUND *csound, MYFLT instr, int skip);
void csoundRemoveInstrumentBreakpoint (CSOUND *csound, MYFLT instr);
void csoundClearBreakpoints (CSOUND *csound);
void csoundSetBreakpointCallback (CSOUND *csound, breakpoint_cb_t bkpt_cb, void *userdata);
void csoundDebugContinue (CSOUND *csound);
void csoundDebugStop (CSOUND *csound);
debug_instr_t *csoundDebugGetInstrInstances(CSOUND *csound);
void csoundDebugFreeInstrInstances(CSOUND *csound, debug_instr_t *instr);
```

```
debug_variable_t *csoundDebugGetVariables(CSOUND *csound, debug_instr_t *instr);
void csoundDebugFreeVariables(CSOUND *csound, debug_variable_t *varHead);
```

- Windows:
  - Soundfonts in Windows had an internal alignment problem which is fixed

## Release Notes for Csound 6.02

This new version has a large number of bug fixes (including clearing all general tickets on SourceForge). It also introduces some major new facilities such as use as a server, code to run Csound in a browser and a large generalisation of filter opcodes to have parameters changeable at audio rate.

- New opcodes:
  - *nstance* opcode schedules a new instrument instance, storing the instance handle.
  - *turnoff* nw variant to stop a given instrument instance.
  - *strfromurl* to set a string from a URL.
- Orchestra:
  - If building supports it, a `#include` string can be a URL or a file.
  - A space is again permitted between a function name and the opening bracket for all functions allowed in Csound5 (but not in general).
  - The Csound command can start with an empty CSD in daemon mode (`--daemon`): do not exit if CSD/orchestra is not given, is empty or does not compile).
- Score:
  - If building supports it, a `#include` string can be a URL or a file.
- Modified Opcodes and Gens:
  - Many filters generalised to allow k- or a-rate parameters. In particular it includes these:

areson	atonex
butterworth filters	fofilter
lowres	lowresx
lpf18	mode
moogladder	moogvcf
reson	resonr
resonx	resonz
statevar	tonex

- The maximum number of presets in `sfont` increased to 16384.
- *cpsmidinn* is now more accurate.
- *max\_k* now behaves like the documentation. There were cases when it gave strange results.

- The *vst4cs* opcodes have been re-factored. FLTK code has been encapsulated. The build system has been updated for Csound 6.
- In *alwayson* opcode changes for better handling of pfields, more reliable insert of an instrument instance for repeating or re-started score sections.
- The signal flow graph opcodes have replaced OpenMP multi-threading with pthreads, using one-time initialization of static structures.
- Frontends:
  - PNaCl is now supported as a platform, allowing Csound to run under the Chrome browser in all enabled operating systems.
- Bugs fixed:
  - *adsynt2* opcode fixed.
  - *fgentmp* opcode fixed.
  - *dates* opcode fixed.
  - fixed a bug in *pvsfilter*.
  - fixed stereo out in *temposcal* and *mincer*.
  - *pan2* opcode fixed.
  - index overflow in *randh* and *randi* fixed.
  - A number of fixes to CsoundVST: initialization, score handling, and MIDI driver initialization, so it now works for Csound 6.
  - fixed *pycalln* for no inputs.
  - fixed/revised setting and use of *ksmps* and *kr* in UDOs.
  - fixed problem in sending a score event from max to csound via csound~ (Ticket #58).
  - If *itype* in *chn\_k* was set to 3 and values are set less than 1, Csound6 used to give an INIT Error. (Ticket #67).
  - A number of reported seg faults have been dealt with.
  - *xtratim* opcode was using incorrect *ekr* value from csound instead of from instance; when used in conjunction with *setksmps*, was causing notes to have very long *xtratim* set and thus notes were effectively not getting turned off
- System changes:
  - A server mode is now available, accepting input via UDP (with --port option).
  - A longstanding bug in extract was detected and fixed. It does suggest that this facility is little used!
  - The way the external score generator was coded is substantially changed. In particular this should fix a very strange bug in Windows.

- Fixed crashing bug with invaline channel callback due to wrong data object being pulled from csound host data.
- Fixed bug in UDOs with no local ksmps where kcounter was being used incorrectly.
- Better checking in channels.
- (Experimental) If the environment variable CS\_UDO\_DIR is set then any files in the directory that have an .udo extension are automatically included at the start of the orchestra. This facility needs review to see if it is what is required.
- (Experimental) There are new cuda GPGPU opcodes (source only): cudasynt (3 versions for additive synthesis, additive synthesis of fsigs and phase vocoder resynthesis) and cudanal (a GPGPU version of pvsanal).
- Internal changes:
  - Many attempts at faster code.
  - Type inference and parsing still improving.
- iOS:
  - Fixed crash where no csoundSetHostImplementedMIDIIO is used on iOS and no \_RTMIDI value is set.
- OSX:
  - Fixed input device name for auhal.

## Release Notes for Csound6

Csound6 is a significant rewrite of much of the code. In particular the API is not compatible, although all orc/sco/csd works should still run.

There are new facilities, like sample accuracy and realtime mode, described below.

IMPORTANT: The environment variable to find plugins are called `OPCODE6DIR64` or `OPCODE6DIR` (note the 6) so it can co-exist with Csound5.

Similarly `.csoundrc` is renamed `.csound6rc`.

Arrays are now mainstream, with syntax and opcode support. They also exist in multidimensional format. They are created (usually) with *init* opcode or *fillarray*.

```
k1[] init 4
```

generates a k-rate 1-D array of length 4. Similarly

```
a2[][] init 4, 4
```

creates a square 4x4 a-rate array.

```
k2[] fillarray 1, 2, 3, 4
```

creates a 4-element vector filled with 1,..4, which also defines the length.

Elements are used via indexing in [] such as `k1[2]` or `a2[2][3]`. One dimensional arrays replace `tvars`, and can be used in opcodes like *maxtab*, *mintab* and *sumtab* (see below). Array setting can be done in left-hand side of opcodes, i.e.:

```
aSigs[0] vco2 .1, 440
aSigs[1] vco2 .1, 880
```

The new realtime priority mode can be switched on with by passing the `--realtime` or setting the `CSOUND_PARAMS` field `realtime_mode` to 1. This has the following effects:

1. all opcode audio file reading/writing is handled asynchronously by a separate thread.
2. all init-pass operations are also performed asynchronously.

Multicore support is totally rewritten using a different algorithm for task-dispatch, which should use less memory and fewer locks.

- New opcodes:
  - *faustgen*
  - *array* -- many new or revised opcodes -- see *Array Opcodes*.
  - *compileorc* takes a filename containing a collection of instrument definitions and compiles them, replacing existing versions. It returns 0 on success.
  - *compilestr* is like *compileorc* but takes a string.
  - *readscore* runs the score preprocessor on a string and then schedules new events via the RT event mechanism, returning 0 if successful.
- Orchestra
  - Note events can start and end in mid-kcycle. As this is an incompatible change it is only invoked when the command-line option `--sample-accurate` is specified. Note that this does not work for tied notes, and use of skipping initialisation has questionable use.
  - Instruments can run at local *ksmps* values using `setksmps iksmps` as in Csound 5 UDOS.
  - Compilation can be done at any stage, new instruments are added or replace old ones. Running instances of old instrument definitions are not affected. Only limitation is that header constants in `instr 0` are read only once at the time of the first compilation. Init-time code can be placed outside instruments in the global space, and this will be executed once-only following the compilation. In this case, score event generation can be completely replaced by orchestra code. See also new opcodes *compileorc* and *compilestr*.
  - New syntax operators `+=`, `-=`, `*=` and `/=`. These are more than syntactic sugar; please use `+=` and `-=` for accumulating reverbs as it gives better multicore behaviour.
  - The opcodes *add*, *sub*, *mul* and *div* have been deleted; use the forms `+ - * /`. Not many people were aware of these opcodes.
  - Any opcode with a single output or with no outputs can be used as a function. Some opcodes might require type annotation to resolve ambiguities, more details on the *Function syntax in Csound 6*.
  - A statement can be broken across lines after a `,` `=` or arithmetic operation.

- There are a range of new or recoded operations on k-valued arrays, most restricted to 1 dimensional arrays (vectors):

```

kans minarray ktab      returns the smallest value in the
                        (possibly) multidimensional array
kans maxarray ktab      is like mintab
kabs sumarray ktab      returns sum of all values in the array
ktab genarray imin, imax[, inc]
                        generates vector of values from imin
                        to imax by increments of inc (default 1)
ktab2 maparray ktab1, "sin" maps the k-rate 1-arg function in
                        the string to every element of the vector
ktab2 maparray_i ktab1, "sin" maps the i-rate 1-arg function
                        in the string to every element of the vector
ktab2 slicearray ktab1, istart, iend
                        returns a slice of ktab1 from ktab1[istart]
                        to ktab1[iend]
copyf2array ktab, kfn    copies data from an ftable to a vector
copya2ftab ktab, kfn    copies data from a vector to an ftable

```

Arithmetic on arrays is allowed. In particular addition, subtraction, multiplication, division on a element-by-element version is provided in arithmetic format. Similar operations between an array and a scalar are allowed.

- Each instance of any instrument has a scratchpad of 4 values that persist; allows values to carry to next use of the instrument; hope it may be useful in legato etc.
- If a table number is given as -1 then an internal sine wave equivalent to `f. 0 16382 10 1` is used. Attempts to write to this table will give unpredictable results, but is not policed. The 16382 can be change by command line option `--sine-size=#` where the # is rounded up to a power of two.
- A number of oscil opcodes now have the f-table parameter as optional, defaulting to the internal sine wave. (*oscill*, *oscilli*, *oscil*, *oscil3*, *oscili*, *foscil*, *foscill*, *loscil*, *loscil3*).
- Score:
  - Score lines can have multiple strings.
  - Change to escape characters in score strings -- they do not happen.
  - Also note the *readscore* opcode.
- Modified Opcodes and Gens:
  - The *k()* function can take an a-rate argument in which case it is a call to *downsamp*.
- Utilities
  - *Hetro/adsyn* analysis files can be machine byte-order independent if created with `-X`. Down side is a longer file and a little slower loading. The *het\_export* utility will create the independent format from the old, and *het\_import* is no longer necessary.
  - *cvanal* and *lpanal* will produce machine independent files if `-X` option is used. The *convolve* and *lpread* etc opcodes will accept either format. You are encouraged to use the machine independent form. Analysis files produced with `-X` can be used on other systems.
- Frontends
- Bugs fixed:

- System Changes:
  - In Linux and OSX the treatment of locales is now thread-safe and local.
- Platform Changes:
- API:

New API functions...

- new configuration/parameter setting functions

```
PUBLIC int csoundSetOption(CSOUND *csound, char *option);
PUBLIC void csoundSetParams(CSOUND *csound, CSOUND_PARAMS *p);
PUBLIC void csoundGetParams(CSOUND *csound, CSOUND_PARAMS *p);
PUBLIC void csoundSetOutput(CSOUND *csound, char *name, char *type,
                             char *format);
PUBLIC void csoundSetInput(CSOUND *csound, char *name);
PUBLIC void csoundSetMIDIInput(CSOUND *csound, char *name);
PUBLIC void csoundSetMIDIFileInput(CSOUND *csound, char *name);
PUBLIC void csoundSetMIDIOutput(CSOUND *csound, char *name);
PUBLIC void csoundSetMIDIFileOutput(CSOUND *csound, char *name);
```

- new parsing/compilation functions

```
PUBLIC TREE *csoundParseOrc(CSOUND *csound, char *str);
PUBLIC int csoundCompileTree(CSOUND *csound, TREE *root);
PUBLIC int csoundCompileOrc(CSOUND *csound, const char *str);
PUBLIC int csoundReadScore(CSOUND *csound, char *str);
PUBLIC int csoundCompileArgs(CSOUND *, int argc, char **argv);
```

- new function for starting csound after first compilation

```
PUBLIC int csoundStart(CSOUND *csound);
```

- new software bus threadsafe getters/setters

```
PUBLIC MYFLT csoundGetControlChannel(CSOUND *csound, const char *name);
PUBLIC void csoundSetControlChannel(CSOUND *csound, const char *name, MYFLT val);
PUBLIC void csoundGetAudioChannel(CSOUND *csound, const char *name, MYFLT *samples);
PUBLIC void csoundSetAudioChannel(CSOUND *csound, const char *name, MYFLT *samples);
PUBLIC void csoundSetStringChannel(CSOUND *csound, const char *name, char *string);
PUBLIC void csoundGetStringChannel(CSOUND *csound, const char *name, char *string);
```

- new table threadsafe copy functions

```
PUBLIC void csoundTableCopyOut(CSOUND *csound, int table, MYFLT *dest);
PUBLIC void csoundTableCopyIn(CSOUND *csound, int table, MYFLT *src);
```

API has been made threadsafe so that performance and control can occur in separate threads (after a call to *csoundStart()* or *csoundCompile()*). Threadsafety is ensure by

1. use of atomic read/writing to control channels
2. spinlocks in audio and string channels
3. mutexes protecting compilation, score events and table access.

- Internal:

- The build system is now cmake (and not scon as in Csound5).
- A number of table access opcodes have been rewritten but should behave the same. Similarly *diskin* and *diskin2* now use the same code and so *diskin* should be more stable.
- The old parser is completely removed.
- New internal functions in Csound

```
void (*FlushCircularBuffer)(CSOUND *, void *);
void (*FileOpenAsync)(CSOUND *, void *, int, const char *, void *,
                     const char *, int, int, int);
unsigned int (*ReadAsync)(CSOUND *, void *, MYFLT *, int);
unsigned int (*WriteAsync)(CSOUND *, void *, MYFLT *, int);
int (*FSeekAsync)(CSOUND *, void *, int, int);
char *(*GetString)(CSOUND *, MYFLT);
    Extract a string originating from a score-event argument.
```

- Functions removed

```
void (*FileOpen)(CSOUND *, void*, int, const char*, void*, const char*);
```

- The "private" parts of the API have been changed considerably. Also structures like EVTBLK have changed.
- The LINKAGE1/FLINKAGE1 macros are renamed as LINKAGE\_BUILTIN/FLINKAGE\_BUILTIN.
- Template for a-rate perf-pass opcodes is

```
int perf_myopcode(CSOUND *csound, MYOPCODE *p)
{
    uint32_t offset = p->h.insdshead->ksmps_offset;
    uint32_t early = p->h.insdshead->ksmps_no_end;
    uint32_t nsmps = CS_KSMPS;
    ...
    if (UNLIKELY(offset)) memset(p->res, '\0', offset*sizeof(MYFLT));
    if (UNLIKELY(early)) {
        nsmps -= early;
        memset(&p->res[nsmps], '\0', early*sizeof(MYFLT));
    }
    for (n=offset; n<nsmps; n++) {
        .....
        p->res[n] = ....
    }
    return OK;
}
```

- String variables re-implemented
- OENTRY structure has changed and has a new dependency field; please use this field as it is required for multicore semantics. You could set it to -1 and disallow all parallelism, but at least it is safe.
- All opcodes that touch audio should take note of sample-accurate code.
- A number of previous API functions are removed; *OpenFile* and *OpenFile2* both replaced by new *OpenFile2* with additional argument.
- Additions have been made for arg type specifications for opcodes.
  - Any-types have been added, as follows:

- '!' signifies a required arg of any-type
- '?' signifies an optional arg of any-type
- '\*' signifies a var-arg list of any-type
- Arrays are now specified using "[x]" where x is a type-specifier. The type-specifier can be any of the of the current specifiers, including any-types. See *Opcodes/arrays.c* for example usage.
- New Type System

A new type system has been added to Csound6, and significant changes have been made to the compiler. The previous system for handling types involved depending on the first-letter of a variable's name every time it was used to determine type. This meant there was a lot of re-checking of types. Also, adding new types was difficult, as there was a lot of custom code that had to be updated to check for new type letters.

In Csound6, a separate system of types was added. Types are defined as CS\_TYPE's. The creation of variables from types and the initialisation of memory has been encapsulated within the CS\_TYPE's. This change allows easier addition of new types, as well as generic calculations of memory pools, amongst other things.

The compiler has been modified since Csound5 to now use the type system as an integral part of its semantic checking phase. Variables are now registered into a CS\_VAR\_POOL when they are first defined, with the CS\_VARIABLE having a reference to its CS\_TYPE. After first time definition within the pool, the type information is then looked up in consequent variable lookups, rather than re-calculated from the variable name. This opens up possibilities for new variable naming and typing strategies, i.e. using "myVar:K" to denote a k-rate arg. This also opens up possibilities for user-defined types, such as "data myType kval, aval", then using "myVar:myType" to define a var of that type. (The previous is speculative, and is not an active proposal at this time.)

The addition of the type system has formalised the static type system that has existed in Csound prior to Csound6. It has, arguably, simplified the code-base in terms of type handling, as well as laid the ground work for future type-related research to be integrated into Csound.

## New in Version 5.19 (2013 January 7)

This is mainly a bug-fixing release but with a number of new opcodes and enhanced features.

- New opcodes:
  - *ipmidi* module for MIDI over network.
  - *ppltrack* opcode.
  - *combinv* opcode.
- New Gen and Macros:
  - Better checking in *GEN28*.
  - Check range in *outrg*, and optionally allow wrapping.
- Orchestra:
  - Change empty statement to a warning.

- Added line numbers to many input args message (new parser).
  - Modified Opcodes and GENs:
    - Better error and warning messages.
    - *loopseg* now checks argument count.
    - *harmon2/3/4* improved.
    - *active*: added the option to skip the instances in release phase.
    - New and more tested implementation of ChordSpace.
  - Bug fixed:
    - Fix botched optimisation in lowpass filters.
    - Chn opcodes fixed in Linux.
    - Fix bug in *loscil* with silence.
    - Correct *GEN23* when comment does not end in newline.
    - Correcting *loopseg*.
    - Number of input and output channels fixed in new parser.
    - Fixed *GEN43* issue.
    - Fixed *fout*.
    - *centroid* was likely to crash.
    - Minor bug in printing which lost %.
    - An uninitialised value in *fold* fixed.
    - Uninitialised values in *dconv* fixed.
    - Assignment of fsigs now works.
  - System Changes:
    - Avoid seg fault on some user errors.
    - Faster modal4 opcodes.
    - Allow cabbage compilation.
    - Made pfield size dynamic in event message csoundapi~.
    - The default output format with pipe and double float is AU.
    - Change to ircam with default format, '-o stdout' and pipe.
- 
- Added double float precision for output format.

- Platform Changes:
  - Linux:
    - Spinlocks initialised (fixes bug in chn opcodes).
  - OSX:
    - Improved selection of devices in rtauhal module.
    - Added a circular buffer interface and lock-free operation to rtauhal.
    - Fixed MacOSX installer (creating symlinks to lib\_csnd.dylib).
  - Haiku:
    - New platform
  - Android:
    - Using `-B` now in android to set circular buffer size.
    - Added fluid synth opcodes for android.
    - Added `inputMessage` method to `CsoundObj`.
    - Allow `CSDPlayer` to be installed on SD Card.
  - iOS:
    - Improved audio routing.
    - Bottom speaker the default for iOS.
  - API:
    - Added new API function `csoundCompileFromStrings()`.

## New in Version 5.18 (2012 August 29)

This is mainly a bug-fixing release but with a number of new opcodes and enhanced features.

- New opcodes:
  - *centroid* opcode like *pvscent* but acting on audio signals
  - *cosseg* like *linseg* but with cosine interpolation
  - *cossegb* like *linsegb* but with cosine interpolation
  - *cossegr* like *linsegr* but with cosine interpolation
  - *joystick* to read input values from an external joystick (Linux only)
  - *log2* function for logarithms base 2
- *platerev* opcode to model a reverberating square plate

- *pwd* opcode to determine the current working directory
- *readf* opcode to read strings from a file
- *readfi* opcode to read strings from a file on initialisation
- *vbap* opcode like other vbap family but flexible about number of speakers and choice of layouts.
- *vbapg* opcode like vbap but only calculate the gains on the channels.
- New functionality
  - Changes to <CsOptions> to allow spaces between words, and escaped characters.
  - *fout* and *fin* use a better buffering strategy, and so are faster
  - It is possible to specify just an orchestra with the --orc flag. This is useful when a score is not needed.
  - A new command-line flag --ogg flag has been added for easy use of ogg/vorbis output.
  - Added alsaseq real-time midi
- Bug fixes and improvements:
  - *dates* opcode could crash on 64bit architecture; this is fixed
  - Some multicore interlocks were wrong. It is believed that this was not actually a problem, but would be in the future.
  - There were cases when a file was double closed, leading to a crash on exit.
  - Two new features added in *partikkel*. Panning law for channelmasks can now be set using a function table (second optional argument to partikkel) and new support opcodes partikkelget and partikkelset, to access and modify the internal mask indices of partikkel.
  - *follow2* was reworked so the i-rate and k-rate calculations are the same.
  - *pvscent* is corrected as it returned half the correct value.
  - *vbaplsinit* can create more than one speaker layout which vbap/vbapg can use. Also much better diagnostics on incorrect layouts.
- Internal Changes:
  - Code changed so bison 2.6 can be used.
  - It is assumed that libsndfile version 1.0.19 or later is available.
  - If the score is omitted a near-infinite wait is generated.

## New in Version 5.17 (March 2012)

This is mainly a bug-fixing release with no major changes, but the number of fixes warrants a release.

- New opcodes:
  - *cell* opcode, for cellular automata

- Modified Opcodes and Gens:
  - active now will report total number of active or allocated instruments if argument in zero
  - stsend and strecv the TCP socket opcodes reworked to a logical design
  - DSSI system now will take up to 9 channels
  - FLsavesnap works with other widgets where imin > imax
- Utilities:
  - csbeats better documented and built by default; also more note lengths available
  - Some security holes in utilities fixed
- Bug fixes:
  - unibrand opcode at a-rate fixed
  - Locale fix for floating point literals in orchestra
  - transegr fixed
- System Changes:
  - Score can now last longer (change to size of time variable)
  - An empty score gives a very long performance time (years and years)
  - Android code released
  - Changes to use of tmp files; now all are deleted at end of run (previously some were left) and the environment variable TMPDIR is used.
  - interaction between Comments, end of line and end of file fixed
  - Hexadecimal numbers now allowed in orchestra
  - Empty orchestra now not a crash
  - change to macro expansion inside a string
  - avoid infinite loop when eof in malformed score macro
  - fixed macroname-with-args diagnostics and memory leak
  - change to preprocessor: { { } } inside "..." and better diagnostics
  - fix windows installer so it removes full \$INSTDIR\bin from PATH during uninstall: this cleans up the PATH environment variable when uninstalling on Windows. Previously, it was leaving a trailing "\bin" on the PATH.
  - CsoundAC MusicModel class more usable by C++ programs
  - ftcps had been missed as a function

- Many! Some messages quietened, code improvements etc

## New in Version 5.16 (February 2012)

The major change is that the new parser is now the default. The old parser is still available in case of difficulty but the new has been given extensive testing since the start of the year, including complete restructuring of macro expansion. A side effect is that the runtime of most orchestras is faster, although parsing is slower. There are a few optimisations implemented like constant folding in simple cases. Line numbers and file names are traced better than before.

Some memory leaks also fixed.

- New opcodes:
  - Opcodes adapted from SuperCollider by Tito Latini: *dust*, *dust2*, *gausstrig*, *gendy*, *gendyc*, and *gendyx*.
  - Fractal noise generator by Tito Latini: *fractalnoise*.
  - Opcodes for accessing table values by direct indexing, by John ffitich: *ptable*, *ptablei*, *ptable3*, and *ptablew*. These opcodes are respectively like *table*, *tablei*, *table3*, and *tablew*, but they do not require a power-of-2 table size.
- Modified Opcodes and Gens:
  - There was a fence post problem in tab opcode that could falsely report a reference out of range.
  - GEN15 mis-called gens 13 and 14 internally, using uninitialised values voice amplitude. Problem fixed.
  - fmbell now takes an optional argument to control the sustain time.
  - Change to pvsbasic for tab to table conversions.
  - poscil is now polymorphic, allowing k- or a-rate amplitude and frequency.
  - p() and i() changed when argument at k-rate.
  - gen49 deferred now works.
  - gen23 now available deferred.
- Utilities:
  - Checked for use with the new parser in memory files.
- Frontends:
  - Table access added to csoundapi~ via new get/set methods.
- Bug fixes and improvements:
  - Many in new parser related to precedence and multicore.
  - Better diagnostics when orchestra file/csd is missing.

- csd file: fix CsFileB and CsSampleB.
- Fixed score statement 'n'.
- Fixed bug in diskin2 leading to infinite loop.
- Fixed bug causing crossfade noise in hrtfmmove.
- Fixed unlikely buffer overflows in some utilities.
- Avoid segfault in midicN.
- Bug in mp3in in skip=0 case fixed.
- 'r' score statement fixed with respect to macros.
- sndwarp could segfault.
- System Changes:
  - Preprocessor #if #else #endin working.
  - #includes depth now limited rather than infinite recursion.
  - Really turn off all displays if --nodisplays or -d is used; fixes bug where using -d or --nodisplays would still cause the winFLTK.c csoundModuleInit to setup display callbacks; bug caused with python TK apps and CsoundYield\_FLTK being called.
  - Memory leak in mp3in and mp3len fixed.
- Internal Changes:
  - Very, very, very many! And the new parser...

## **New in Version 5.15 (December 2011)**

- New opcodes:
  - *ftab2tab* opcode.
  - *tab2pvs* opcode.
  - *pvs2tab* opcode.
  - *cpumeter* opcode, (not really new but now available in OSX)
  - *minmax* opcode.
  - *(EXPERIMENTAL) ftresize* opcode.
  - *(EXPERIMENTAL) ftresizei* opcode.
  - *hrtfearly* opcode.
  - *hrtfverb* opcode.

New Gen and Macros

- Code to allow GEN49 to be deferred [NB does not seem to work]
- Modified Opcodes and Gens
  - socksend and sockrecv no longer uses MTFU check and work on Windows
  - mpulse changed so if next event is at negative time use the absolute value
  - serial opcode now runs on Windows as well as Un\*x
  - out, out2, outq, outh, outo outx and out32 are now identical opcodes and will take up to as many arguments as nchnls. This replaces the current remapping of opcodes
  - turnoff2 now polymorphic wrt S and k types (ie accepts instrumnet names)
- Bugs fixed:
  - GEN42 fixed
  - jacko: fixed a segfault removing the unused JackSessionID option
  - doppler memory leak fixed
  - transegr fixed in release mode when skipping most of envelope
  - FLPack now agrees with manual
  - max\_k now agrees with manual
  - hrtfverb fixed
  - atsa code now works on Windows in more cases
  - tabmorph bug fixed
  - fixed problem with user-defined opcodes having no outputs
  - Various fixes to \* ... \*/ comments
- System Changes:
  - Various licence issues sorted
  - Loris is no longer part of the Csound tree
  - Memory leaks fixed
  - If no score is given a dummy that runs for over 100 years is created
  - All score processing takes place in memory without temporary files
  - String memory now expandable and no size limitation
  - #if #else #end now in new parser
  - Adjustments to MIDI file precision in output
  - On OSX move from Coreaudio to AuHAL

- Multicore now safe for ZAK, Channels and modifying tables
- New coremidi module
- Virtual Keyboard improved: 1) Dropdown for choosing base octave (the one that starts with the virtual key mapped to physical key Z). Default value is 5 which is backwards compatible. 2) Shift-X mappings which add two octaves to X mappings for a total of 4 octaves playable from the physical keyboard (starting from selected base octave). 3) Control-N / Control-Shift-N mappings to increment / decrement slider for control N. 4) Mouse wheel now controls sliders.
- tsig type for vectors
- tsigs and fsigs allowed as arguments in UDOs
- API: Minor version upped
- Internal Changes:
  - Very, very, very many!
  - 
  -

## **New in Version 5.14 (October 2011)**

- New opcodes:
  - *mp3len* opcode.
  - *qnan* opcode.
  - *qinf* opcode.
  - *exprandi* opcode.
  - *cauchy* opcode.
  - *gaussi* opcode.
  - *cpumeter* opcode.
  - *linsegb* opcode.
  - *expsegb* opcode.
  - *transegb* opcode.
  - *expsegba* opcode.
  - *pvsgain* opcode.
  - *pvsbufread2* opcode.
  - *serial* opcodes.
  - *lua opcodes* opcodes.

- *plustab* opcode.
  - *multtab* opcode.
  - *maxarray* opcode.
  - *minarray* opcode.
  - *sumarray* opcode.
  - *scalearray* opcode.
  - New functionality
    - beats processor renamed to *csbeats* and distributed
    - *mkdb* utility to provide a catalogue of plugin libraries/opcodes
    - *ladspa* library build in default system
    - macros are now expanded inside string in the score
    - *there in an until .. do .. od* looping syntax (in the new parser only)
    - SIGPIPE signals are ignored rather than causing Csound to exit
    - It is possible to use vectors of k-rate values, named t-variable. They are initialised to a fixed size with *init* and can be read with a simple `[]` syntax. assignment to elements is only via `=`. There are also a few new opcodes that provide wider functionality.
  - Bug fixes and improvements:
    - reading values to fill tables was broken with respect to comments
    - internal error in *wii\_data* fixed
    - *pvsshift* fixed
    - *jacko* fixed
    - *gen23* minor fixes
    - *wiimote* fixed
    - *atsaadd* fixed
    - *compress* fixed to work with *0dbfs*
    - *pvdbufread* corrected with respect to position counting
    - *tempo* opcode fixed
    - *CsFileB* section in *.csd* files had a bug, now fixed
    - deferred *gen01* tables could have wrong size
-

- fixed memory issue in ATSSinoi
- various fixes to cscore
- various fixes to partials and tradsyn
- transegr could crash in some cases
- loris opcodes updated to latest version
- date opcode has new base in some platforms to avoid overflow
- pvsblur now works over reinit
- diskin, diskin2 and soundin now can read up to 40 channels
- prints behaves better with rounding
- fmpercfl now has working vibrato
- atreson now has gain parameter at k-rate
- comb opcode made safe if in and out arguments the same
- better accuracy in line and expon
- OSCsend recovers space previously lost
- OSCsend can send a table as a blob with the T tag -- experimental and untested.
- lpf18 now has an optional iskip argument
- i() will also accept an i-rate value in which case it is a no-op
- makecsd revised and extended to have options for MIDI and score processing and licenses
- lpanal reworked to remove bugs and oddities
- an issue with noise in alsa fixed and a click in portaudio fixed
- portaudio driver changed to be more robust on stop/exit
- Internal Changes:
  - Many many changes to the new parser so it is now operational, but should be used with care
  - The multicore system is distributed in an experimental mode and should be used with great care.
  - 
  -

## **New in Version 5.13 (January 2011)**

- New opcodes:
  - *median* opcode.

- *filevalid* opcode.
- *pvstanal*, *pvs warp*, *temposcal*, *pvslock* spectral processing opcodes.
- *mincer* opcode
- *fareylen* sequence opcodes.
- New functionality
  - Real random number generators using `/dev/random` (Linux only).
  - INF macro added to orchestras; z read as infinity in scores
  - init changed to allow multiple inits in on statement
  - GEN for support of farey sequences
  - *maxalloc*, *cpuprc*, *active* now accept named instruments.
  - If normalisation in pow opcodes is zero treat as 1
  - *inch* can take upto 20 inputs and outputs.
  - *pvscale*, *pvsvoc* and *pvmix* now have very good spectral envelope preservation modes (1 = filtered cepstrum, 2 = true envelope).
  - *oscill* could be static if the duration was long; now there is a positive minimum increment.
  - GEN49 now uses search paths.
- Bug fixes and improvements:
  - Count of lines fixed in orchestras, and \ inside strings
  - Fast tab opcodes made safe from crashes
  - % in formatted printing could crash
  - Double free in fgen fixed
  - *sndwarp* quietened (gave too many messages)
  - gen41 deals with positive probabilities
  - *adsynt* reworked removing many bugs
  - *adsynt2* phase error fixed
  - Bug in max number of gens fixed
  - Better checking in grain4
  - Better checking in *adsyn*
  - modulus was wrong in new parser
  - *atonex/tonex* did wrong operation

- *mp3in* could repeat sound at end of file
- *changed* opcode initialised to zero
- Serious bug in *tabmorpha* fixed
- GEN49 has serious bug removed, so no longer incorrect silences.
- *partikkel* opcode: fixed bug in sub-sample grain placement when using grain rate FM
- Internal Changes:
  - In the new parser only there are operator @ and @@ to round up the next integer to a power of 2 or powerof2+1
  - Score sorting made much faster
  - *lineto* improved
  - Named gens allowed
  - Various printing include instrument name if available
  - Command option to omit loading a library
  - Number of out channels no longer constrained to be number of in
  - Many fixes to new parser
  - More use of Warnings than Messages (allows for them to be switched off)
  - *csoundSetMessageCallback* reset if callback set to null

## New in Version 5.12 (January 2010)

- New opcodes:
  - *transegr* is a version of the *transeg* opcode which has a release section which is triggered by midi, a *turnoff2* opcode or a negative instrument number *i score event*.
  - *ftgenonce* generates a function table from within an instrument definition, without duplication of data.
  - *passign* allows quick initialization of i-rate variables from p-fields
  - *crossfm* implements crossed fm synthesis.
  - *loopxseg* is like *loopseg* but with exponential envelope.
  - *looptseg* is like *loopseg* but with a flexible envelope like *transeg*
- Bug fixes and improvements:
  - *pvshift* would overwrite in double mode.
  - *pan2* case 3 fixed.
  - *clockon* and *clockoff* now work again.

- *cross2* and *interp* could have divided by zero
- *linecount* for error messages no longer includes text from *.csoundrc*
- *p5gconnect* changed to use a separate thread to avoid timeout problem.
- *transeg* checks argument count.
- *sfloat* used to be limited to 10 sound fonts and was not policed. Now open-ended.
- Internal Changes:
  - \" allowed as an escape in orchestral strings
  - New parser fixed on optional arguments
  - Better checking of *f* statement with negative number
  - Soundfonts only initialise pitches array once, in the soundfont opcodes.
  - Usual collection of gratuitous minor changes, layout and comments

## New in Version 5.11 (June 2009)

- New opcodes:
  - *mp3in* allows reading of mp3 files directly in the orchestra.
  - *wiiconnect*, *wiidata*, *wiisend*, *wiirange* opcodes by john ffitc to receive and send data to a wiimote controller.
  - New opcodes to receive data directly from a p5glove by john ffitc *p5gdata*
  - *tabsum* sums sections of ftables
  - *MixerSetLevel\_i* an init-time only version of *MixerSetLevel*
  - *doppler* implements a simulation of the doppler effect.
  - *filebit* reports the file depth of a file.
  - The new *Signal Flow opcodes* enable the usage of signal flow graphs in Csound.
- New functionality
  - New panning type for *pan2* opcode
  - New *csd* score tag <CsExScore>.
  - New *-Ma* option for ALSA RT MIDI module which listens to all devices.
  - There is a *gen49* to read mp3 files
  - Added rounding bin code to *pvscale*
  - Added non-power-of-2 table support for *ftload* and *ftsave*

- GEN23 totally rewritten to be more consistent in what constitutes a separator and comments. (Still no /\* \*/ comments)
- Bug fixes and improvements:
  - New examples for pvs opcodes by Joachim Heintz: pvsarp, pvscent, pvsbandp, pvsbandr, pvsbufread, pvsadsyn, pvsynth, pvsblur, pvscale, pvscross, pvsfilter, pvsfreeze, pvshift, pvsmaska, pvsmorph
  - Use of automatic numbering of ftables reuses table numbers
  - seed with positive argument was wrong
  - sprintf with an empty string printed wrong data
  - mute now works with both numeric and named instruments
  - Small fixes in diskln, and in tablexkt
- Internal Changes:
  - SConstruct now builds completely independent shared libraries for Python, Lua, and Java wrappers.
  - New Parser almost usable
  - Redrawing of graphs fixed so that only selected ones get redrawn.
  - RT-alsa more forgiving on near sample rates
  - It is possible to have the score generated by an external program rather than using standard score format using <CScore bin="translator"> to call the program translator on the score data
  - lpc\_export fixed
  - Removed limit on macro names length
  - PMAX, the number of arguments to a score event has been reduced by 2, and an overflow system introduced so GENs can have arbitrary numbers of arguments.
  - Increased API version to 2.1.
  - New API function pointer ldmemfile2withCB() which is a version of ldmemfile() allowing a callback to be set and called exactly once to process the MEMFIL buffer after it is loaded.
  - csound->floatsize set; zero in earlier versions
  - GetChannelLock added

## **New in Version 5.10 (December 2008)**

- New functionality
  - New option to listen to all MIDI devices using the portmidi realtime module. To enable listening to all devices use "-+rtmidi=portmidi -Ma".
  - Dither on output implemented; rectangular and triangular dither available in some cases
  - GEN20 type 6 now has option to set variance

- Bug fixes and improvements:
  - Locale set to C numeric to avoid , versus . problems.
  - *diskin* fixed
  - *outo* was broken regarding channel 6
  - *pitchamdf* fixed
  - *zfilter2* initialization fixed
  - *s32b14* fixed
  - Fixed other bugs fixed that have not been reported publicly.
- Internal Changes:
  - The major version of the Csound API is increased to 2; affected *csound.so* as well. This means that Csound 5.10 is incompatible with applications ("front ends", "clients", or "hosts") that were built for Csound 5.08 and earlier and that use API version 1.x. These applications will need to be rebuilt to work with the current and future versions of Csound. Csound front ends written in interpreted languages such as Python or Java may continue to work without modification. It may also be possible to keep both an earlier version of the Csound library and an API 2.0 version on the same machine together so that new and old Csound-based applications can run side-by-side. These changes do not in any way affect the compatibility of Csound orchestras and scores: all old documents should continue to work as before.
  - Time now counted internally in samples, overcoming a longstanding bug with rounding of time to k-rate.
  - Many internal changes related to branch prediction. Some opcodes are substantially quicker.
  -

## New in Version 5.09 (October 2008)

- New opcodes:
  - New *vosim* opcode by Rasmus Ekman which recreates the historic VOSIM (VOcal SIMulator) technique.
  - New *dcblock2* opcode by Victor Lazzarini.
  - New Chua's oscillator model: *chuap* by Michael Gogins.
  - New *Linear Algebra* opcodes by Michael Gogins. Standard Linear algebra over real and complex vectors and matrices: elementwise arithmetic, norms, transpose and conjugate, inner products, matrix inverse, LU decomposition, QR decomposition, and QR-based eigenvalue decomposition. Includes copying vectors to and from a-rate signals, function tables, and f-signals.
  - New ambisonic opcodes: *bformdec1* and *bformenc1*. These opcodes deprecate the older *bformdec* and *bformenc*.
  - New Score control opcodes by Victor Lazzarini: *rewindscore* and *setscorepos*.
- New functionality:

- The *vbap* family of opcodes (*vbap4*, *vbap8*, *vbap16* and *vbapz*) now accept k-rate variables for all their input arguments.
- New pulseaudio I/O module on Linux.
- New optional *ienv* parameter to generate envelopes for the soundfont opcodes: *sfplay*, *sfplay3*, *sfplaym* and *sfplay3m*.
- Added 'skip normalisation argument' to "tanh" named GEN routine. (See *Named GEN Routines*)
- Added scheduler priority option on alsa.
- Bug fixes and improvements:
  - Allow scientific notation (as was in csound4!) in *GEN23*.
  - Fixed bug in FLTK initialization. Should make FLTK usage more stable.
  - Error on /\* \*/ comments in orchestra fixed.
  - *poscil* no longer overwrites frequency if variable is shared.
  - *printk* and *printks* check that opcode is initialised.
  - Deprecate *soundout* and *soundouts* in favour of *fout*.
  - Fixed *space* opcode to accept non-pow-2 (deferred) tables.
  - Fixed *pvsmorph* bug.
- Internal Changes:
  - New parser has #include and argumentless macros.
  - Less casting between floats and doubles in float version.
  - Includes experimental multicore support.
  - *buzz* opcode rewritten.
  - Many other internal changes and small bug fixes.

## New in Version 5.08 (February 2008)

- New opcodes:
  - *imagecreate*, *imagesize*, *imagegetpixel*, *imagesetpixel*, *imagesave*, *imageload* and *imagefree*: New image file processing opcodes by Cesare Marilungo to read/write png images from Csound.
  - *pvsbandp* and *pvsbandr* by John ffitch, which perform band-pass and band-reject filtering in the spectral domain on a pvs signal.
  - New HRTF opcodes by Brian Carty: *hrtfmove*, *hrtfmove2* and *hrtfstat*.
  - New waveshaping opcodes: *powershape*, *polynomial*, *chebyshevpoly*, *pdclip*, *pdhalf*, *pdhalfy*, and *syncephasor*

- New jack transport control opcode: *jacktransport*
- New functionality
  - Added `--csd-line-nums=` command line option to select mode for error line reporting.
  - New "no-carry" operator (!) for score language that prevents implicit carrying of p-fields in i-statements.
  - Added `--syntax-check-only` commandline flag (exclusive with `--i-only`)
  - `<CsLicence>` tag for *CSDs*. `<CsLicense>` is accepted as an alternative to `<CsLicence>`.
- Bug fixes and improvements:
  - Changed order of outputs for *hilbert*. This change breaks compatibility with previous versions, but fixes the opcode and now works as documented.
  - Messages about loading opcode plugins modified so can be suppressed with message level flag.
  - Major changes to score error reporting; now accurately reports the line numbers for the chain of inputs for most errors.
  - Corrected *pan2* so it agrees with documentation.
  - `<CsVersion>` tag works again according to the manual.
  - Fixed the { and } score looping statements. Added missing documentation for them and ~, &, |, and # operators in score expressions.
  - *hilbert* had its outputs reversed, now correct. Manual example updated.
- Internal Changes:
  - Change to gettext localisation; French and Columbian-Spanish translations available.
  - Internal changes to *partikkel*, interpolation of waveform read and windowing, allowing more precise pitch synchronous granular synthesis. Updated examples for *partikkel*.
  - *pvscale*: Improved algorithm for SDFT case so no amplitude variation.

## New in Version 5.07 (October 2007)

- New opcodes:
  - *pan2*: a stereo panning opcode
  - *cpsmidinn*, *pchmidinn*, *octmidinn*: converters for MIDI note numbers
  - *fluidSetInterpMethod*: interpolation in fluid sound fonts
  - *sflooper*: a soundfont version of *flooper2*
  - *pvsbuffer* and *pvsbufread*: buffering/reading of fsigs for delays/timescale changes.
- New functionality

- SDFT - the Sliding Discrete Fourier Transform -- added seamlessly to *pvsanal*, etc opcodes if the overlap is less than the *k*smpls or less than 10. Some *pvsXXX* opcodes extended to take a-rate parameters when sliding.
- New feature (*-O null / --logfile=null*) that disables all messages and printing to the console.
- Bug fixes and improvements:
  - *partikkel* -- particle synthesis had an inadvertent bug, now fixed.
  - Closing of MIDI input on Windows(MM) failed; now fixed
  - *fluidEngine* opcode now takes optional number of channels (range 16-256, default to 256) and polyphony (range 16-4096, default to 4096) to use.
  - *atsa* utility safer when given silence.
  - *ATSaddnz*: improved checking.
  - Ambisonics (*bformdec*, *bformenc*) has more options for controlled opposites.
  - Bug in *turnoff2* fixed.
  - *het\_export*: invalid check caused export to fail.
- Internal Changes:
  - Improved Windows installer.
  - CsoundVST replaced by CsoundAC, that does not depend on the VST SDK headers.
  - Less messages in Windows(MM) startup.
  - P argument type added (*k*-rate defaults to 1) for opcode in and out types.

## New in Version 5.06 (June 2007)

- New granular opcodes: *partikkel*, *partikkelsync* and *diskgrain*.
- New opcode for event dispatch: *scoreline*.
- Many new opcodes from Gabriel Maldonado's CsoundAV: *hvs1*, *hvs2*, *hvs3*, *vphaseseg*, *inrg*, *outrg*, *lposcila*, *lposcilsa*, *lposcilsa2*, *tabmorph*, *tabmorpha*, *tabmorphi*, *tabmorphak*, *trandom*, *vtable1k*, *slider8table*, *slider16table*, *slider32table*, *slider64table*, *slider8tablef*, *slider16tablef*, *slider32tablef*, *slider64tablef*, *sliderKawai* and the a-rate version of *ctrl7*.
- Also from CsoundAV, many new FLTK widget opcodes: *FLkeyIn*, *FLslidBnk2*, *FLvslidBnk*, *FLvslidBnk2*, *FLmouse*, *FLxyin*, *FLhvsBox*, *FLslidBnkSet*, *FLslidBnkSetk*, *FLslidBnk2Set*, *FLslidBnk2Setk*, *FLslidBnkGetHandle*,
- New *pvs* opcodes: *pvsdiskin*, *pvsmorph*,
- *eqfil*
- New command line options (*--m-warnings*) to control messages
- *csladspa*: a CSD to LADSPA plugin kit.

- And many bug fixes including (but not limited to): fixed k-rate version of *system*; fixed scaling problems of *vrandh* and *vrandi*; fixed occasional failure of *turnoff*; fixed OS X bug; fixed *ATScross* and fixed *mod*.

Csound5GUI now works properly on all platforms and csoundapi~ (pd object) has been updated.

---

# Part I. Overview

---

---

# Table of Contents

Introduction .....	4
The Csound Command .....	5
Order of Precedence .....	5
Description of the command syntax .....	5
Csound command line .....	7
Command-line Flags (by Category) .....	17
Csound Environment Variables .....	28
Unified File Format for Orchestras and Scores .....	31
Description .....	31
Example .....	33
Command Line Parameter File (.csoundrc) .....	34
Score File Preprocessing .....	34
The Extract Feature .....	34
Independent Pre-Processing with Scsort .....	35
Using Csound .....	36
Csound's Console Output .....	36
How Csound works .....	37
Amplitude values in Csound .....	38
Real-Time Audio .....	40
Realtime I/O on Linux .....	40
Mac OSX .....	46
Windows .....	47
Realtime I/O with JACK Connection Kit .....	48
Optimizing Audio I/O Latency .....	49
Configuring .....	51
Syntax of the Orchestra .....	52
Orchestra Header Statements .....	52
Instrument and Opcode Block Statements .....	53
Ordinary Statements .....	54
Types, Constants and Variables .....	54
Variable Initialization .....	55
Expressions .....	55
Directories and Files .....	56
Nomenclature .....	56
Macros .....	57
Named Instruments .....	57
User Defined Opcodes (UDO) .....	60
K-Rate Vectors .....	60
Function Syntax in Csound6 .....	60
The Standard Numeric Score .....	62
Preprocessing of Standard Scores .....	62
Carry .....	62
Tempo .....	63
Sort .....	63
Score Statements .....	64
Next-P and Previous-P Symbols .....	64
Ramping .....	65
Score Macros .....	66
Multiple File Score .....	68
Evaluation of Expressions .....	68
Strings in p-fields .....	70

Front Ends .....	71
CsoundAC .....	72
CsoundVST .....	74
Building Csound .....	76
Csound Links .....	77

---

# Introduction

Csound is a unit generator-based, user-programmable computer music system. It was originally written by Barry Vercoe at the Massachusetts Institute of Technology in 1984 as the first C language version of this type of software. Since then Csound has received numerous contributions from researchers, programmers, and musicians from around the world.

Around 1991, John ffitich ported Csound to Microsoft DOS. Csound currently runs on many varieties of UNIX and Linux, Microsoft DOS and Windows, all versions of the Macintosh operating system including Mac OS X, and others.

There are newer computer music systems that have graphical patch editors (e.g. Max/MSP, PD, jMax, or Open Sound World), or that use more advanced techniques of software engineering (e.g. Nyquist or SuperCollider). Yet Csound still has the largest and most varied set of unit generators, is the best documented, runs on the most platforms, and is the easiest to extend. It is possible to compile Csound using double-precision arithmetic throughout for superior sound quality. In short, Csound must be considered one of the most powerful musical instruments ever created.

In addition to this "canonical" version of Csound and CsoundAC, there are other versions of Csound and other front ends for Csound, many of which can be found at <http://csound.github.io>.

---

# The Csound Command

The command *csound* is a basic frontend to the system that can be used to generate a sound output from an *orchestra* file and a *score* file (or a unified *csd file*). It is designed to be called from a terminal or DOS window. In addition to it, there are other *front-ends*, which might be simpler to use. The score file can be in one of many different formats, according to user preference. Translation, sorting, and formatting into orchestra-readable numeric text is handled by various preprocessors; all or part of the score is then sent on to the orchestra. Orchestra performance is influenced by *command flags*, which set the level of displays and console reports, specify I/O filenames and sample formats, and declare the nature of real-time sensing and control.

## Order of Precedence

There are five places where options for Csound performance may be set. They are processed in the following order:

1. Csound's own defaults
2. File defined by the CSOUNDRC *environment variable*, or .csoundrc file in the HOME directory
3. A .csoundrc file in the current directory
4. <CsOptions> tag in a .csd file
5. Passed on the Csound *command line*

The later options in the list will override any earlier ones. As of version 5.01 of Csound, sample and control rate override flags (*-r* and *-k*) specified anywhere override sr, kr, and ksmps defined in the orchestra header.

## Description of the command syntax

The *csound* command is followed by a set of *Command Line Flags* and the name of the orchestra (*.orc*) and score (*.sco*) files or the *Unified csd file* (containing both orchestra and score) to process. *Command Line Flags* to control input and output configuration may appear anywhere in the command line, either separately or bundled together. A flag taking a Name or Number will find it in that argument, or in the immediately subsequent one. The following are thus equivalent commands:

```
csound -nm3 orchname -Sxxfilename scorename  
csound -n -m 3 orchname -x xfilename -S scorename
```

All flags and names are optional. The default values are:

```
csound -s -otest -b1024 -B1024 -m7 -P128 orchname scorename
```

where *orchname* is a file containing Csound orchestra code, and *scorename* is a file of score data in standard numeric score format, optionally presorted and time-warped. If *scorename* is omitted, there are two default options:

1. if real-time input is expected (e.g. *-L*, *-M*, *-iadc* or *-F*), a dummy score file is substituted consisting of the single statement 'f 0 3600' (i.e. listen for RT input for one hour)
2. else Csound uses the previously processed *score.srt* in the current directory.

Csound reports on the various stages of score and orchestra processing as it executes, performing various syntax and error checks along the way. Once the actual performance has begun, any error messages will derive from either the instrument loader or the unit generators themselves. A CSound command may include any rational combination of flag arguments.

## Running the examples in this manual from the command line

Most of the manual's examples come ready to run without the need of adding any command line flags since they specify options within the csd file's <CsOptions> tag, so you only need to type something like:

```
csound oscil.csd
```

within the examples folder, and realtime audio output should be generated.

# Csound command line

csound

## Description

The *csound* command executes Csound.

## Syntax

```
csound [flags] [orchname] [scorename]
```

```
csound [flags] [csdfilename]
```

## Csound command line flags

Listed below are the command line flags available in Csound6 in alphabetical order. Various platform implementations may not react the same way to different flags! You can view the command line flags organized by category in *Command-line Flags (by Category)*.

The command line arguments are of 2 types: *flags* arguments (beginning with a “-”, “--” or “-+”), and *name* arguments (such as filenames). Certain flag arguments take a following name or numeric argument. Flags that start with “--” and “-+” usually take an argument themselves using “=”.

### Command-line Flags

-@ FILE	Provide an extended command-line in file “FILE”
-3, --format=24bit	Use 24-bit audio samples.
-8, --format=uchar	Use 8-bit unsigned character audio samples.
--format= <i>type</i>	Set the audio file output format to one of the formats available in libsndfile. At present the list is aiff, au, avr, caf, flac, htk, ircam, mat4, mat5, MPC, nist, ogg, paf, pvf, raw, sd2, sds, svx, voc, w64, W64, wav, wavex, WVE, xi. Can also be used as --format=type:format or --format=format:type to set both the file type (wav, aiff, etc.) and sample format (short, long, float, etc.) at the same time.
-A, --aiff, --format=aiff	Write an AIFF format soundfile. Use with the <i>-c</i> , <i>-s</i> , <i>-l</i> , or <i>-f</i> flags.
-a, --format=alaw	Use a-law audio samples.
-B NUM, --hardwarebufsamps=NUM	Number of audio sample-frames held in the DAC <i>hardware</i> buffer. This is a threshold on which <i>software</i> audio I/O (above) will wait before returning. A small number reduces audio I/O delay; but the value is often hardware limited, and small values will risk data lates. In the case of portaudio output (the default real-time output), the <i>-B</i> parameter (more precisely, <i>-B / sr</i> ) is passed as the “suggested latency” value. Other than that, Csound has no control over how

	PortAudio interprets the parameter. The default is 1024 on Linux, 4096 on Mac OS X and 16384 on Windows.
-b NUM, --iobufsamps=NUM	Number of audio sample-frames per sound i/o <i>software</i> buffer. Large is efficient, but small will reduce audio I/O delay and improve the accuracy of the timing of real time events. The default is 256 on Linux, 1024 on MacOS X, and 4096 on Windows. In real-time performance, Csound waits on audio I/O on <i>NUM</i> boundaries. It also processes audio (and polls for other input like MIDI) on orchestra <i>ksmps</i> boundaries. The two can be made synchronous. For convenience, if NUM is negative, the effective value is <i>ksmps</i> * <i>-NUM</i> (audio synchronous with k-period boundaries). With NUM small (e.g. 1) polling is then frequent and also locked to fixed DAC sample boundaries.
	Note: if both <i>-iadc</i> and <i>-odac</i> are used at the same time (full duplex real time audio), the <i>-b</i> option should be set to an integer multiple of <i>ksmps</i> .
-C, --cscore	Use Cscore processing of the scorefile.
-c, --format=schar	Use 8-bit signed character audio samples.
--csd-line-nums=NUM	Determines how line numbers are counted and displayed for error messages when processing a Csound Unified Document file (.csd). This flag has no effect if separate orchestra and score files are used. (Csound 5.08 and later).
	<ul style="list-style-type: none"> <li>• 0 = line numbers are relative to the beginning of the orchestra or score sections of the CSD</li> <li>• 1 = line numbers are relative to the beginning of the CSD file. This is the default as of Csound 5.08.</li> </ul>
-D, --defer-gen1	Defer GEN01 soundfile loads until performance time.
-d, --nodisplays	Suppress all displays. See <i>-O</i> if you want to save the log to a file.
	<div style="display: flex; align-items: center;">  <div> <p><b>Note</b></p> <p>This option only works from the command line and not from .csound6rc or a CsOptions section of a csd file. It needs to be acted upon before sound proper starts. It is ignored in .csoundrc and CsOptions.</p> </div> </div>
-d	Run in daemon mode: do not exit if CSD/orchestra is not given, is empty or does not compile.
--devices	Gives a list of available audio devices and then exits.
--devices[=x]	list audio devices (x=out, output devices only; x=in, input; else input and output) and exit.
--displays	Enables displays, reverting the effect of any previous <i>-d</i> flag.
--default-paths	Reenables adding of directory of CSD/ORC/SCO to search paths, if it has been disabled by a previous <i>--no-default-paths</i> (e.g. in .csoundrc).

`--env:NAME=VALUE` Set environment variable `NAME` to `VALUE`. Note: not all environment variables can be set this way, because some are read before parsing the command line. `INCDIR`, `SADIR`, `SFDIR`, and `SSDIR` are known to work.

`--env:NAME+=VALUE` Append `VALUE` to ';' separated list of search paths in environment variable `NAME` (should be `INCDIR`, `SADIR`, `SFDIR`, or `SSDIR`). If a file is found in multiple directories, the last will be used.

`--expression-opt` *Note that this option has no affect in csound6. In Csound 5 only.* Turns on some optimizations in expressions:

- Redundant assignment operations are eliminated whenever possible. This means that for example this line `a1 = a2 + a3` will compile as `a1 Add a2, a3` instead of `#a0 Add a2, a3 a1 = #a0` saving a temporary variable and an opcode call. Less opcode calls result in reduced CPU usage (an average orchestra may compile about 10% faster with `--expression-opt`, but it depends largely on how many expressions are used, what the control rate is (see also below), etc.; thus, the difference may be less, but also much more).
- number of a- and k-rate temporary variables is significantly reduced. This expression

```
(a1 + a2 + a3 + a4)
```

will compile as

```
#a0 Add a1, a2
#a0 Add #a0, a3
#a0 Add #a0, a4 ; (the result is in #a0)
```

instead of

```
#a0 Add a1, a2
#a1 Add #a0, a3
#a2 Add #a1, a4 ; (the result is in #a2)
```

The advantages of less temporary variables are:

- less cache memory is used, which may improve performance of orchestras with many a-rate expressions and a low control rate (e.g. `ksmps = 100`)
- large orchestras may load faster due to less different identifier names
- index overflow errors (i.e. when messages like this `Case2: indx=-56004 (ffff253c); (short)indx = 9532 (253c)` are printed and odd behavior or a Csound crash occurs) may be fixed, because such errors are triggered by too many different (especially a-rate) variable names in a single instrument.

Note that this optimization (due to technical reasons) is not performed on i-rate temporary variables.



## Warning

When `--expression-opt` is turned on, it is not allowed to use the `i()` function with an expression argument, and relying on the value of k-rate expressions at i-time is unsafe.

-F FILE, --midifile=FILE	Read MIDI events from MIDI file <i>FILE</i> . The file should have only one track in Csound versions 4.xx and earlier; this limitation is removed in Csound 5.00.
-f, --format=float	Use single-format float audio samples (not playable on some systems, but can be read by <i>-i</i> , <i>soundin</i> and <i>GEN01</i>
-G, --postscriptdisplay	Suppress graphics, use PostScript displays instead.
-g, --asciidisplay	Suppress graphics, use ASCII displays instead.
--get-system-sr	prints system sr and exits, requires previous <code>-o dac</code> . If the audio does not support this request then <code>-1</code> is reported.
-H#, --heartbeat=NUM	Print a heartbeat after each soundfile buffer write: <ul style="list-style-type: none"> <li>• no NUM, a rotating bar.</li> <li>• NUM = 1, a rotating bar.</li> <li>• NUM = 2, a dot (.)</li> <li>• NUM = 3, filesize in seconds.</li> <li>• NUM = 4, sound a bell.</li> </ul>
-h, --noheader	No header on output soundfile. Don't write a file header, just binary samples.
--help	Display on-line help message.
-I, --i-only	<i>i-time only</i> . Allocate and initialize all instruments as per the score, but skip all p-time processing (no k-signals or a-signals, and thus no amplitudes and no sound). Provides a fast validity check of the score pfields and orchestra i-variables. This option is exclusive of the <code>--syntax-check-only</code> flag.
-i FILE, --input=FILE	Input soundfile name. If not a full pathname, the file will be sought first in the current directory, then in that given by the environment variable <i>SSDIR</i> (if defined), then by <i>SFDIR</i> . The name <i>stdin</i> will cause audio to be read from standard input. <p>The name <i>devaudio</i> or <i>adc</i> will request sound from the host audio input device. It is possible to select a device number by appending an integer value in the range 0 to 1023, or a device name separated by a <code>:</code> character (e.g. <code>-iadc3</code>, <code>-iadc:hw:1,1</code>). It depends on the host audio interface whether a device number or a name should be used. In the first case, an out of range number usually results in an error and listing the valid device numbers.</p>

	The audio coming in using <i>-i</i> can be received using opcodes like <i>inch</i> .
<code>--id_artist=string</code>	(max. length = 200 characters) Artist tag in output soundfile (no spaces)
<code>--id_comment=string</code>	(max. length = 200 characters) Comment tag in output soundfile (no spaces)
<code>--id_copyright=string</code>	(max. length = 200 characters) Copyright tag in output soundfile (no spaces)
<code>--id_copyright=integer</code>	(Since version 6.05) Simple copyright/licence encoded as an integer. Coding is:  0: "All rights reserved" (default) 1: "Creative Commons Attribution-NonCommercial-NoDerivatives (CC BY-NC-ND)" 2: "Creative Commons Attribution-NonCommercial-ShareAlike (CC BY-NC-SA)" 3: "Creative Commons Attribution-NonCommercial (CC BY-NC)" 4: "Creative Commons Attribution-NoDerivatives (CC BY-ND)" 5: "Creative Commons Attribution-ShareAlike (CC BY-SA)" 6: "Creative Commons Attribution-ShareAlike (CC BY)" 7: "Licenced under BSD"
<code>--id_date=string</code>	.
	(max. length = 200 characters) Date tag in output soundfile (no spaces)
<code>--id_software=string</code>	(max. length = 200 characters) Software tag in output soundfile (no spaces)
<code>--id_title=string</code>	(max. length = 200 characters) Title tag in output soundfile (no spaces)
<code>--ignore_csopts=integer</code>	If set to 1, Csound will ignore all options specified in the csd file's CsOptions section. See <i>Unified File Format for Orchestras and Scores</i> .
<code>--input_stream=string</code>	Pulseaudio input stream name.
<code>-J, --ircam, --format=ircam</code>	Write an IRCAM format soundfile.
<code>-j NUM</code>	Make <i>NUM</i> processes available for rendering. This is only advantageous if the number of processors on the computer is the same or more than the number of requested processes. It also may slow rendering down if ksmps is too small.
<code>--jack_client=[client_name]</code>	The client name used by Csound, defaults to 'csound5'. If multiple instances of Csound connect to the JACK server, different client names need to be used to avoid name conflicts. (Linux and Mac OS X only)

--jack\_inportname=[input port name prefix], --jack\_outportname=[output port name prefix]

Name prefix of Csound JACK input/output ports; the default is 'input' and 'output'. The actual port name is the channel number appended to the name prefix. (Linux and Mac OS X only)

Example: with the above default settings, a stereo orchestra will create these ports in full duplex operation:

```
csound5:input1      (record left)
csound5:input2      (record right)
csound5:output1     (playback left)
csound5:output2     (playback right)
```

-K, --nopeaks

Do not generate any PEAK chunks.

-k NUM, --control-rate=NUM

Override the control rate (*KR*) supplied by the orchestra.

-L DEVICE, --score-in=DEVICE

Read line-oriented real-time score events from device *DEVICE*. The name *stdin* will permit score events to be typed at your terminal, or piped from another process. Each line-event is terminated by a carriage-return. Events are coded just like those in a *standard numeric score*, except that an event with *p2=0* will be performed immediately, and an event with *p2=T* will be performed *T* seconds after arrival. Events can arrive at any time, and in any order. The score *carry* feature is legal here, as are held notes (*p3* negative) and string arguments, but ramps and *pp* or *np* references are not.



### Note

The -L flag is only valid on \*NIX systems which have pipes. It doesn't work on Windows.

-l, --format=long

Use long integer audio samples.

-M DEVICE, --midi-device=DEVICE

Read MIDI events from device *DEVICE*. If using ALSA MIDI (*-+rtmidi=alsa*), devices are selected by name and not number. So, you need to use an option like *-M hw:CARD,DEVICE* where *CARD* and *DEVICE* are the card and device numbers (e.g. *-M hw:1,0*). In the case of PortMidi and MME, *DEVICE* should be a number, and if it is out of range, an error occurs and the valid device numbers are printed. When using PortMidi, you can use '-Ma' to enable all devices. This is also convenient when you don't have devices as it will not generate an error.

-m NUM, --messagelevel=NUM

Message level for standard (terminal) output. Takes the *sum* of any of the following values:

- 1 = note amplitude messages
- 2 = samples out of range message
- 4 = warning messages
- 128 = print benchmark information

And exactly one of these to select note amplitude format:

- 0 = raw amplitudes, no colours

	<ul style="list-style-type: none"><li>• 32 = dB, no colors</li><li>• 64 = dB, out of range highlighted with red</li><li>• 96 = dB, all colors</li><li>• 256 = raw, out of range highlighted with red</li><li>• 512 = raw, all colours</li></ul> <p>The default is 135 (128+4+2+1), which means all messages, raw amplitude values, and printing elapsed time at the end of performance. The coloring of raw amplitudes was introduced in version 5.04.</p>
<code>--m-amps=NUM</code>	<p>Message level for amplitudes on standard (terminal) output.</p> <ul style="list-style-type: none"><li>• 0 = no note amplitude messages</li><li>• 1 = note amplitude messages</li></ul>
<code>--m-range=NUM</code>	<p>Message level for out of range messages on standard (terminal) output.</p> <ul style="list-style-type: none"><li>• 0 = no samples out of range message</li><li>• 1 = samples out of range message</li></ul>
<code>--m-warnings=NUM</code>	<p>Message level for warnings on standard (terminal) output.</p> <ul style="list-style-type: none"><li>• 0 = no warning messages</li><li>• 1 = warning messages</li></ul>
<code>--m-dB=NUM</code>	<p>Message level for amplitude format on standard (terminal) output.</p> <ul style="list-style-type: none"><li>• 0 = absolute amplitude messages</li><li>• 1 = dB amplitude messages</li></ul>
<code>--m-colours=NUM</code>	<p>Message level for amplitude format on standard (terminal) output.</p> <ul style="list-style-type: none"><li>• 0 = no colouring of amplitude messages</li><li>• 1 = colouring of amplitude messages</li></ul>
<code>--m-benchmarks=NUM</code>	<p>Message level for benchmark information on standard (terminal) output.</p> <ul style="list-style-type: none"><li>• 0 = no benchmark numbers</li><li>• 1 = print benchmark numbers</li></ul>
<code>--max_str_len=integer</code>	<p>(min: 10, max: 10000) Maximum length of string variables + 1; defaults to 256 allowing a length of 255 characters. The length of string constants is not limited by this parameter.</p>
<code>--midi-devices[=x]</code>	<p>list midi devices (x=out, output devices only; x=in, input; else input and output) and exit.</p>

<code>--midi-key=N</code>	Route MIDI note on message key number to pfield N as MIDI value [0-127].
<code>--midi-key-cps=N</code>	Route MIDI note on message key number to pfield N as cycles per second.
<code>--midi-key-oct=N</code>	Route MIDI note on message key number to pfield N as linear octave.
<code>--midi-key-pch=N</code>	Route MIDI note on message key number to pfield N as oct.pch (pitch class).
<code>--midi-velocity=N</code>	Route MIDI note on message velocity number to pfield N as MIDI value [0-127].
<code>--midi-velocity-amp=N</code>	Route MIDI note on message velocity number to pfield N as amplitude [0-0dbFS].
<code>--midioutfile=FILENAME</code>	Save MIDI output to a file (Csound 5.00 and later only).
<code>++msg_color=boolean</code>	Enable message attributes (colors etc.); might need to be disabled on some terminals which print strange characters instead of modifying text attributes. default: true.
<code>++mute_tracks=string</code>	(max. length = 255 characters) Ignore events (other than tempo changes) in MIDI file tracks defined by pattern (for example, <code>++mute_tracks=00101</code> will mute the third and fifth tracks).
<code>-N, --notify</code>	Notify (ring the bell) when score or MIDI track is done.
<code>-n, --nosound</code>	No sound. Do all processing, but bypass writing of sound to disk. This flag does not change the execution in any other way.
<code>--num-threads=NUM</code>	Make <i>NUM</i> processes available for rendering. This is only advantageous if the number of processors on the computer is more than the number of requested processes. It also may slow rendering down if <i>ksmps</i> is too small.
<code>--no-default-paths</code>	Disables adding of directory of CSD/ORC/SCO to search paths.
<code>--no-expression-opt</code>	Disables expression optimization.
<code>-O FILE, --logfile=FILE</code>	Log output to file <i>FILE</i> . If <i>FILE</i> is null (i.e. <code>-O null</code> or <code>--logfile=null</code> ) all printing of messages to the console is disabled.



### Note

This option only works from the command line and not from `.csound6rc` or a `CsOptions` section of a `csd` file. It needs to be acted upon before sound proper starts. It is ignored in `.csoundrc` and `CsOptions`.

<code>-o FILE, --output=FILE</code>	Output soundfile name. If not a full pathname, the soundfile will be placed in the directory given by the environment variable <i>SFDIR</i> (if defined), else in the current directory. The name <i>stdout</i> will cause audio to be written to standard output, while <i>null</i> results in no sound output similarly to the <code>-n</code> flag. If no name is given, the default name will be <i>test</i> .
-------------------------------------	--

The name *devaudio* or *dac* (you can use *-odac* or *-o dac*) will request writing sound to the host audio output device. It is possible to select a device number by appending an integer value in the range 0 to 1023, or a device name separated by a `:` character (e.g. *-odac3*, *-odac:hw:1,1*). It depends on the host audio interface whether a device number or a name should be used. In the first case, an out of range number usually results in an error and listing the valid device numbers.

<code>--ogg</code>	Set output file format to ogg. (csound 5.18 and later)
<code>--omacro:XXX=YYY</code>	Set orchestra macro <code>XXX</code> to value <code>YYY</code>
<code>--opcode-lib=LIBNAME</code>	Load plugin library <i>LIBNAME</i> .
<code>--orc orcname</code>	Set the argument as the orchestra file. Used when not score is required>. (Csound 5.18 and later).
<code>--ksmps=N</code>	Set ksmps override to <code>N</code> (6.05 and later).
<code>++output_stream=string</code>	Pulseaudio output stream name.
<code>--port=N</code>	Set UDP port on which to listen instruments/orchestra code (implies <code>--daemon</code> )
<code>-Q DEVICE</code>	<p>Enables MIDI OUT operations to device id <i>DEVICE</i>. This flag allows parallel MIDI OUT and DAC performance. Unfortunately the real-time timing implemented in Csound is completely managed by DAC buffer sample flow. So MIDI OUT operations can present some time irregularities. These irregularities can be reduced by using a lower value for the <i>-b</i> flag.</p> <p>If using ALSA MIDI (<code>++rtmidi=alsa</code>), devices are selected by name and not number. So, you need to use an option like <code>-Q hw:CARD,DEVICE</code> where <i>CARD</i> and <i>DEVICE</i> are the card and device numbers (e.g. <code>-Q hw:1,0</code>). In the case of PortMidi and MME, <i>DEVICE</i> should be a number, and if it is out of range, an error occurs and the valid device numbers are printed.</p>
<code>-R, --rewrite</code>	Continually rewrite the header while writing the soundfile (WAV/AIFF).
<code>-r NUM, --sample-rate=NUM</code>	Override the sampling rate ( <i>SR</i> ) supplied by the orchestra.
<code>++raw_controller_mode=boolean</code>	Disable special handling of MIDI controllers like sustain pedal, all notes off etc., allowing the use of all the 128 controllers for any purpose. This will also set the initial value of all controllers to zero. Default: no.
<code>--realtime</code>	<p>realtime priority mode is switched on which the following effects:</p> <ol style="list-style-type: none"> <li>1. all opcode audio file reading/writing is handled asynchronously by a separate thread.</li> <li>2. all init-pass operations are also performed asynchronously.</li> </ol>

<code>--rtaudio=string</code>	(max. length = 20 characters) Real time audio module name. The default is PortAudio. Also available, depending on platform and build options: Linux: alsa, jack; Windows: mme; Mac OS X: CoreAudio. In addition, null can be used on all platforms, to disable the use of any real time audio plugin.
<code>--rtmidi=string</code>	(max. length = 20 characters) Real time MIDI module name. Defaults to PortMidi, other options (depending on build options): Linux: alsa; Windows: mme, winmm. In addition, null can be used on all platforms, to disable the use of any real time MIDI plugin.  ALSA MIDI devices are selected by name and not number. So, you need to use an option like <code>-M hw:CARD,DEVICE</code> where CARD and DEVICE are the card and device numbers (e.g. <code>-M hw:1,0</code> ).
<code>-s, --format=short</code>	Use short integer audio samples.
<code>--sample-accurate</code>	Start and stop instances of instruments at the nearest sample to the requested time. This is in contrast to traditional Csound which rounds the times to the nearest k-cycle. Note that this does not work with tied notes.
<code>--sched</code>	<i>Linux only.</i> Use real-time scheduling and lock memory. (Also requires <code>-d</code> and either <code>-o dac</code> or <code>-o devaudio</code> ). See also <code>--sched=N</code> below.
<code>--sched=N</code>	<i>Linux only.</i> Same as <code>--sched</code> , but allows specifying a priority value: if N is positive (in the range 1 to 99) the scheduling policy SCHED_RR will be used with a priority of N; otherwise, SCHED_OTHER is used with the nice level set to N. Can also be used in the format <code>--sched=N,MAXCPU,TIME</code> to enable the use of a "watchdog" thread that terminates Csound if the average CPU usage exceeds MAXCPU percents over a period of TIME seconds (new in Csound 5.00).
<code>--server=string</code>	Pulseaudio server name.
<code>--skip_seconds=float</code>	(min: 0) Start playback at the specified time (in seconds), skipping earlier events in the score and MIDI file.
<code>--smacro:XXX=YYY</code>	Set score macro XXX to value YYY
<code>--strset</code>	<i>Csound 5.</i> The <code>--strset</code> option allows setting strset string values from the command line, in the format <code>'--strsetN=VALUE'</code> . It is useful for passing parameters to the orchestra (e.g. file names).
<code>--syntax-check-only</code>	Causes Csound to exit immediately after the orchestra and score parsers finish checking the syntax of the input files and before the orchestra performs the score. This option is exclusive of the <code>--i-only</code> flag. (Csound 5.08 and later).
<code>-T, --terminate-on-midi</code>	Terminate the performance when the end of MIDI file is reached.
<code>-t0, --keep-sorted-score</code>	Prevents Csound from deleting the sorted score file, score.srt, upon exit.

-t NUM, --tempo=NUM	Use the uninterpreted beats of <i>score.srt</i> for this performance, and set the initial tempo at <i>NUM</i> beats per minute. When this flag is set, the tempo of score performance is also controllable from within the orchestra. <b>WARNING:</b> this mode of operation is experimental and may be unreliable.
-U UTILITY, --utility=UTILITY	Invoke the utility program <i>UTILITY</i> . Use any invalid name to list the available utilities.
-u, --format=ulaw	Use u-law audio samples.
--vbr-quality=X	Set variable bit-rate quality for output to ogg. (Csound 6.03 and later).
-v, --verbose	Verbose translate and run. Prints details of orch translation and performance, enabling errors to be more clearly located.
--version	Exits after printing version information.
-W, --wave, --format=wave	Write a WAV format soundfile.
-x FILE, --extract-score=FILE	Extract a portion of the sorted score, <i>score.srt</i> , using the extract file <i>FILE</i> (see <i>Extract</i> ).
-Z, --dither	Switch on dithering of audio conversion from internal floating point to 32, 16 and 8-bit formats. The default form of the dither is triangular.
-Z, --dither--triangular, --dither--uniform	Switch on dithering of audio conversion from internal floating point to 32, 16 and 8-bit formats. In the case of <i>-Z</i> the next digit should be a 1 (for triangular) or a 2 (for uniform). The exact interpretation depends on the output system.
-z NUM, --list-opcodesNUM	List opcodes in this version: <ul style="list-style-type: none"> <li>• no NUM, just show names</li> <li>• NUM = 0, just show names</li> <li>• NUM = 1, show arguments to each opcode using the format &lt;op-name&gt; &lt;outargs&gt; &lt;inargs&gt;</li> <li>• NUM = 2, show names including deprecated ones</li> <li>• NUM = 3, show arguments to each opcode, including deprecated ones, using the format &lt;opname&gt; &lt;outargs&gt; &lt;inargs&gt;</li> </ul>

## Command-line Flags (by Category)

Listed below are the command line available in Csound5 organized by categories. Various platform implementations may not react the same way to different flags!

You can view the command line flags organized alphabetically in *Command-line Flags (Alphabetically)*.

The format of a command is either:

```
csound [flags] [orchname] [scorename]
```

or

`csound [flags] [csdfilename]`

where the arguments are of 2 types: *flags* arguments (beginning with a “-”, “--” or “+”), and *name* arguments (such as filenames). Certain flag arguments take a following name or numeric argument. Flags that start with “--” and “+” usually take an argument themselves using “=”.

## Audio File Output

<code>-3, --format=24bit</code>	Use 24-bit audio samples.
<code>-8, --format=uchar</code>	Use 8-bit unsigned character audio samples.
<code>-A, --aiff, --format=aiff</code>	Write an AIFF format soundfile. Use with the <code>-c</code> , <code>-s</code> , <code>-l</code> , or <code>-f</code> flags.
<code>-a, --format=alaw</code>	Use a-law audio samples.
<code>-c, --format=schar</code>	Use 8-bit signed character audio samples.
<code>-f, --format=float</code>	Use single-format float audio samples (not playable on some systems, but can be read by <code>-i</code> , <code>soundin</code> and <code>GEN01</code>
<code>--format=type</code>	Set the audio file output format to one of the formats available in <code>libsndfile</code> . At present the list is <code>aiff</code> , <code>au</code> , <code>avr</code> , <code>caf</code> , <code>flac</code> , <code>htk</code> , <code>ircam</code> , <code>mat4</code> , <code>mat5</code> , <code>nis</code> , <code>paf</code> , <code>pvf</code> , <code>raw</code> , <code>sd2</code> , <code>sds</code> , <code>svx</code> , <code>voc</code> , <code>w64</code> , <code>wav</code> , <code>wavex</code> and <code>xi</code> . Can also be used as <code>--format=type:format</code> or <code>--format=format:type</code> to set both the file type ( <code>wav</code> , <code>aiff</code> , etc.) and sample format ( <code>short</code> , <code>long</code> , <code>float</code> , etc.) at the same time.
<code>-h, --noheader</code>	No header on output soundfile. Don't write a file header, just binary samples.
<code>-i FILE, --input=FILE</code>	Input soundfile name. If not a full pathname, the file will be sought first in the current directory, then in that given by the environment variable <code>SSDIR</code> (if defined), then by <code>SFDIR</code> . The name <code>stdin</code> will cause audio to be read from standard input.  The name <code>devaudio</code> or <code>adc</code> will request sound from the host audio input device. It is possible to select a device number by appending an integer value in the range 0 to 1023, or a device name separated by a <code>:</code> character. It depends on the host audio interface whether a device number or a name should be used. In the first case, an out of range number usually results in an error and listing the valid device numbers.  The audio coming in using <code>-i</code> can be received using opcodes like <code>inch</code> .
<code>-J, --ircam, --format=ircam</code>	Write an IRCAM format soundfile.
<code>-K, --nopeaks</code>	Do not generate any PEAK chunks.
<code>-l, --format=long</code>	Use long integer audio samples.
<code>-n, --nosound</code>	No sound. Do all processing, but bypass writing of sound to disk. This flag does not change the execution in any other way.

-o FILE, --output=FILE	Output soundfile name. If not a full pathname, the soundfile will be placed in the directory given by the environment variable <i>SFDIR</i> (if defined), else in the current directory. The name <i>stdout</i> will cause audio to be written to standard output, while <i>null</i> results in no sound output similarly to the -n flag. If no name is given, the default name will be <i>test</i> .  The name <i>dac</i> or <i>devaudio</i> (you can use <i>-odac</i> or <i>-o dac</i> ) will request writing sound to the host audio output device. It is possible to select a device number by appending an integer value in the range 0 to 1023, or a device name separated by a : character. It depends on the host audio interface whether a device number or a name should be used. In the first case, an out of range number usually results in an error and listing the valid device numbers.
--ogg	Set output file format to ogg. (Csound 5.18 and later).
--vbr-quality=X	Set variable bit-rate quality for output to ogg. (Csound 6.03 and later).
-R, --rewrite	Continually rewrite the header while writing the soundfile (WAV/AIFF).
-s, --format=short	Use short integer audio samples.
-u, --format=ulaw	Use u-law audio samples.
-W, --wave, --format=wave	Write a WAV format soundfile.
-Z, --dither	Switch on dithering of audio conversion from internal floating point to 32, 16 and 8-bit formats. The default form of the dither is triangular.
-Z, --dither--triangular, --dither--uniform	Switch on dithering of audio conversion from internal floating point to 32, 16 and 8-bit formats. In the case of -Z the next digit should be a 1 (for triangular) or a 2 (for uniform). The exact interpretation depends on the output system.

### Output File Id tags

--id_artist=string	(max. length = 200 characters) Artist tag in output soundfile (no spaces)
--id_comment=string	(max. length = 200 characters) Comment tag in output soundfile (no spaces)
--id_copyright=string	(max. length = 200 characters) Copyright tag in output soundfile (no spaces)
--id_scopyright=integer	(Single version 6.05) Simple copyright/licence encoded as an integer. Coding is:  0: "All rights reserved" (default) 1: "Creative Commons Attribution-NonCommercial-NoDerivatives (CC BY-NC-ND)" 2: "Creative Commons Attribution-NonCommercial-ShareAlike (CC BY-NC-SA)"

- 3: "Creative Commons Attribution-NonCommercial (CC BY-NC)"
- 4: "Creative Commons Attribution-NoDerivatives (CC BY-ND)"
- 5: "Creative Commons Attribution-ShareAlike (CC BY-SA)"
- 6: "Creative Commons Attribution-ShareAlike (CC BY)"
- 7: "Licenced under BSD"

--id_date=string	(max. length = 200 characters) Date tag in output soundfile (no spaces)
--id_software=string	(max. length = 200 characters) Software tag in output soundfile (no spaces)
--id_title=string	(max. length = 200 characters) Title tag in output soundfile (no spaces)

### Realtime Audio Input/Output

-i adc[DEVICE], --input=adc[DEVICE]	The name <i>devaudio</i> or <i>adc</i> will request sound from the host audio input device. It is possible to select a device number by appending an integer value in the range 0 to 1023, or a device name separated by a : character (e.g. -iadc3, -iadc:hw:1,1). It depends on the host audio interface whether a device number or a name should be used. In the first case, an out of range number usually results in an error and listing the valid device numbers.
-o dac[DEVICE], --output=dac[DEVICE]	The name <i>dac</i> or <i>devaudio</i> (you can use <i>-odac</i> or <i>-o dac</i> ) will request writing sound to the host audio output device. It is possible to select a device number by appending an integer value in the range 0 to 1023, or a device name separated by a : character (e.g. -odac3, -odac:hw:1,1). It depends on the host audio interface whether a device number or a name should be used. In the first case, an out of range number usually results in an error and listing the valid device numbers.
--rtaudio=string	(max. length = 20 characters) Real time audio module name. The default is PortAudio (all platforms). Also available, depending on platform and build options: Linux: alsa, jack; Windows: mme; Mac OS X: CoreAudio. In addition, null can be used on all platforms, to disable the use of any real time audio plugin.
--realtime	realtime priority mode is switched on which the following effects: <ol style="list-style-type: none"> <li>1. all opcode audio file reading/writing is handled asynchronously by a separate thread.</li> <li>2. all init-pass operations are also performed asynchronously.</li> </ol>
--server=string	Pulseaudio server name.
--output_stream=string	Pulseaudio output stream name.
--input_stream=string	Pulseaudio input stream name.
--jack_client=[client_name]	The client name used by Csound, defaults to 'csound5'. If multiple instances of Csound connect to the JACK server, different client

names need to be used to avoid name conflicts. (Linux and Mac OS X only)

--jack\_inportname=[input port name prefix], --jack\_outportname=[output port name prefix]

Name prefix of Csound JACK input/output ports; the default is 'input' and 'output'. The actual port name is the channel number appended to the name prefix. (Linux and Mac OS X only)

Example: with the above default settings, a stereo orchestra will create these ports in full duplex operation:

```
csound5:input1      (record left)
csound5:input2      (record right)
csound5:output1     (playback left)
csound5:output2     (playback right)
```

### MIDI File Input/Output

--devices[=x] list audio devices (x=out, output devices only; x=in, input; else input and output) and exit.

-F FILE, --midifile=FILE Read MIDI events from MIDI file *FILE*. The file should have only one track in Csound versions 4.xx and earlier; this limitation is removed in Csound 5.00.

--midioutfile=FILENAME Save MIDI output to a file (Csound 5.00 and later only).

--mute\_tracks=string (max. length = 255 characters) Ignore events (other than tempo changes) in MIDI file tracks defined by pattern (for example, --mute\_tracks=00101 will mute the third and fifth tracks).

--raw\_controller\_mode=boolean Disable special handling of MIDI controllers like sustain pedal, all notes off etc., allowing the use of all the 128 controllers for any purpose. This will also set the initial value of all controllers to zero. Default: no.

--skip\_seconds=float (min: 0) Start playback at the specified time (in seconds), skipping earlier events in the score and MIDI file.

-T, --terminate-on-midi Terminate the performance when the end of MIDI file is reached.

### MIDI Realtime Input/Output

-M DEVICE, --midi-device=DEVICE Read MIDI events from device *DEVICE*. If using ALSA MIDI (-+rtmidi=alsa), devices are selected by name and not number. So, you need to use an option like -M hw:CARD,DEVICE where CARD and DEVICE are the card and device numbers (e.g. -M hw:1,0). In the case of PortMidi and MME, DEVICE should be a number, and if it is out of range, an error occurs and the valid device numbers are printed. When using PortMidi, you can use '-Ma' to enable all devices. This is also convenient when you don't have devices as it will not generate an error.

--midi-key=N Route MIDI note on message key number to pfield N as MIDI value [0-127].

--midi-key-cps=N Route MIDI note on message key number to pfield N as cycles per second.

<code>--midi-key-oct=N</code>	Route MIDI note on message key number to pfield N as linear octave.
<code>--midi-key-pch=N</code>	Route MIDI note on message key number to pfield N as oct.pch (pitch class).
<code>--midi-velocity=N</code>	Route MIDI note on message velocity number to pfield N as MIDI value [0-127].
<code>--midi-velocity-amp=N</code>	Route MIDI note on message velocity number to pfield N as amplitude [0-0dbFS].
<code>--midioutfile=FILENAME</code>	Save MIDI output to a file (Csound 5.00 and later only).
<code>--rtmidi=string</code>	(max. length = 20 characters) Real time MIDI module name. Defaults to PortMidi, other options (depending on build options): Linux: <code>alsa</code> ; Windows: <code>mme</code> , <code>winmm</code> . In addition, null can be used on all platforms, to disable the use of any real time MIDI plugin.  ALSA MIDI devices are selected by name and not number. So, you need to use an option like <code>-M hw:CARD,DEVICE</code> where <code>CARD</code> and <code>DEVICE</code> are the card and device numbers (e.g. <code>-M hw:1,0</code> ).
<code>-Q DEVICE</code>	Enables MIDI OUT operations to device id <i>DEVICE</i> . This flag allows parallel MIDI OUT and DAC performance. Unfortunately the real-time timing implemented in Csound is completely managed by DAC buffer sample flow. So MIDI OUT operations can present some time irregularities. These irregularities can be reduced by using a lower value for the <code>-b</code> flag.  If using ALSA MIDI ( <code>--rtmidi=alsa</code> ), devices are selected by name and not number. So, you need to use an option like <code>-Q hw:CARD,DEVICE</code> where <code>CARD</code> and <code>DEVICE</code> are the card and device numbers (e.g. <code>-Q hw:1,0</code> ). In the case of PortMidi and MME, <code>DEVICE</code> should be a number, and if it is out of range, an error occurs and the valid device numbers are printed.

## Display

<code>--csd-line-nums=NUM</code>	Determines how line numbers are counted and displayed for error messages when processing a Csound Unified Document file (.csd). This flag has no effect if separate orchestra and score files are used. (Csound 5.08 and later). <ul style="list-style-type: none"> <li>• 0 = line numbers are relative to the beginning of the orchestra or score sections of the CSD</li> <li>• 1 = line numbers are relative to the beginning of the CSD file. This is the default as of Csound 5.08.</li> </ul>
<code>-d, --nodisplays</code>	Suppress all displays. See <code>-O</code> if you want to save the log to a file.



### Note

This option only works from the command line and not from `.csound6rc` or a `CsOptions` section of a `csd`

file. It needs to be acted upon before sound proper starts. It is ignored in `.csoundrc` and `CsOptions`.

<code>--displays</code>	Enables displays, reverting the effect of any previous <code>-d</code> flag.
<code>-G, --postscriptdisplay</code>	Suppress graphics, use PostScript displays instead.
<code>-g, --asciisplay</code>	Suppress graphics, use ASCII displays instead.
<code>-H#, --heartbeat=NUM</code>	Print a heartbeat after each soundfile buffer write: <ul style="list-style-type: none"><li>• no NUM, a rotating bar.</li><li>• NUM = 1, a rotating bar.</li><li>• NUM = 2, a dot (.)</li><li>• NUM = 3, filesize in seconds.</li><li>• NUM = 4, sound a bell.</li></ul>
<code>-m NUM, --messagelevel=NUM</code>	Message level for standard (terminal) output. Takes the <i>sum</i> of any of the following values: <ul style="list-style-type: none"><li>• 1 = note amplitude messages</li><li>• 2 = samples out of range message</li><li>• 4 = warning messages</li><li>• 128 = print benchmark information</li></ul> And exactly one of these to select note amplitude format: <ul style="list-style-type: none"><li>• 0 = raw amplitudes, no colours</li><li>• 32 = dB, no colors</li><li>• 64 = dB, out of range highlighted with red</li><li>• 96 = dB, all colors</li><li>• 256 = raw, out of range highlighted with red</li><li>• 512 = raw, all colours</li></ul> The default is 135 (128+4+2+1), which means all messages, raw amplitude values, and printing elapsed time at the end of performance. The coloring of raw amplitudes was introduced in version 5.04
<code>--m-amps=NUM</code>	Message level for amplitudes on standard (terminal) output. <ul style="list-style-type: none"><li>• 0 = no note amplitude messages</li><li>• 1 = note amplitude messages</li></ul>
<code>--m-range=NUM</code>	Message level for out of range messages on standard (terminal) output. <ul style="list-style-type: none"><li>• 0 = no samples out of range message</li></ul>

	<ul style="list-style-type: none"><li>• 1 = samples out of range message</li></ul>
--m-warnings=NUM	Message level for warnings on standard (terminal) output. <ul style="list-style-type: none"><li>• 0 = no warning messages</li><li>• 1 = warning messages</li></ul>
--m-dB=NUM	Message level for amplitude format on standard (terminal) output. <ul style="list-style-type: none"><li>• 0 = absolute amplitude messages</li><li>• 1 = dB amplitude messages</li></ul>
--m-colours=NUM	Message level for amplitude format on standard (terminal) output. <ul style="list-style-type: none"><li>• 0 = no colouring of amplitude messages</li><li>• 1 = colouring of amplitude messages</li></ul>
--m-benchmarks=NUM	Message level for benchmark information on standard (terminal) output. <ul style="list-style-type: none"><li>• 0 = no benchmark numbers</li><li>• 1 = print benchmark numbers</li></ul>
++msg_color=boolean	Enable message attributes (colors etc.); might need to be disabled on some terminals which print strange characters instead of modifying text attributes. default: true.
-v, --verbose	Verbose translate and run. Prints details of orch translation and performance, enabling errors to be more clearly located.
-z NUM, --list-opcodesNUM	List opcodes in this version: <ul style="list-style-type: none"><li>• no NUM, just show names</li><li>• NUM = 0, just show names</li><li>• NUM = 1, show arguments to each opcode using the format &lt;op-name&gt; &lt;outargs&gt; &lt;inargs&gt;</li><li>• NUM = 2, show names including deprecated ones</li><li>• NUM = 3, show arguments to each opcode, including deprecated ones, using the format &lt;opname&gt; &lt;outargs&gt; &lt;inargs&gt;</li></ul>

## Performance Configuration and Control

-B NUM, --hardwarebufsamps=NUM	Number of audio sample-frames held in the DAC <i>hardware</i> buffer. This is a threshold on which <i>software</i> audio I/O (above) will wait before returning. A small number reduces audio I/O delay; but the value is often hardware limited, and small values will risk data lates. In the case of portaudio output (the default real-time output), the -B parameter (more precisely, -B / sr) is passed as the "suggested
--------------------------------	---

latency" value. Other than that, Csound has no control over how PortAudio interprets the parameter. The default is 1024 on Linux, 4096 on Mac OS X and 16384 on Windows.

**-b NUM, --iobufsamps=NUM** Number of audio sample-frames per sound *i/o software* buffer. Large is efficient, but small will reduce audio I/O delay and improve the accuracy of the timing of real time events. The default is 256 on Linux, 1024 on MacOS X, and 4096 on Windows. In real-time performance, Csound waits on audio I/O on *NUM* boundaries. It also processes audio (and polls for other input like MIDI) on orchestra *ksmps* boundaries. The two can be made synchronous. For convenience, if *NUM* is negative, the effective value is *ksmps* \* *-NUM* (audio synchronous with k-period boundaries). With *NUM* small (e.g. 1) polling is then frequent and also locked to fixed DAC sample boundaries.

Note: if both *-iadc* and *-odac* are used at the same time (full duplex real time audio), the *-b* option should be set to an integer multiple of *ksmps*.

**-d** Run in daemon mode: do not exit if CSD/orchestra is not given, is empty or does not compile.

**-k NUM, --control-rate=NUM** Override the control rate (*KR*) supplied by the orchestra.

**-L DEVICE, --score-in=DEVICE** Read line-oriented real-time score events from device *DEVICE*. The name *stdin* will permit score events to be typed at your terminal, or piped from another process. Each line-event is terminated by a carriage-return. Events are coded just like those in a *standard numeric score*, except that an event with *p2=0* will be performed immediately, and an event with *p2=T* will be performed *T* seconds after arrival. Events can arrive at any time, and in any order. The score *carry* feature is legal here, as are held notes (*p3* negative) and string arguments, but ramps and *pp* or *np* references are not.



**Note**

The *-L* flag is only valid on \*NIX systems which have pipes. It doesn't work on Windows.

**--omacro:XXX=YYY** Set orchestra macro *XXX* to value *YYY*

**--port=N** Set UDP port on which to listen instruments/orchestra code (implies *--daemon*)

**-r NUM, --sample-rate=NUM** Override the sampling rate (*SR*) supplied by the orchestra.

**--sample-accurate** Start and stop instances of instruments at the nearest sample to the requested time. This is in contrast to traditional Csound which rounds the times to the nearest *k-cycle*. Note that this does not work with tied notes.

**--sched** *Linux only.* Use real-time scheduling and lock memory. (Also requires *-d* and either *-o dac* or *-o devaudio*). See also *--sched=N* below.

<code>--sched=N</code>	<i>Linux only.</i> Same as <code>--sched</code> , but allows specifying a priority value: if <i>N</i> is positive (in the range 1 to 99) the scheduling policy <code>SCHED_RR</code> will be used with a priority of <i>N</i> ; otherwise, <code>SCHED_OTHER</code> is used with the nice level set to <i>N</i> . Can also be used in the format <code>--sched=N,MAXCPU,TIME</code> to enable the use of a "watchdog" thread that terminates Csound if the average CPU usage exceeds <code>MAXCPU</code> percents over a period of <code>TIME</code> seconds (new in Csound 5.00).
<code>--smacro:XXX=YYY</code>	Set score macro <code>XXX</code> to value <code>YYY</code>
<code>--strset</code>	<i>Csound 5.</i> The <code>--strset</code> option allows setting <code>strset</code> string values from the command line, in the format <code>'--strsetN=VALUE'</code> . It is useful for passing parameters to the orchestra (e.g. file names).
<code>--skip_seconds=float</code>	(min: 0) Start playback at the specified time (in seconds), skipping earlier events in the score and MIDI file.
<code>-t NUM, --tempo=NUM</code>	Use the uninterpreted beats of <i>score.srt</i> for this performance, and set the initial tempo at <i>NUM</i> beats per minute. When this flag is set, the tempo of score performance is also controllable from within the orchestra. <b>WARNING:</b> this mode of operation is experimental and may be unreliable.
<code>-j NUM, --num-threads=NUM</code>	Make <i>NUM</i> processes available for rendering. This is only advantageous if the number of processors on the computer is the same or more than the number of requested processes. It also may slow rendering down if <code>ksmps</code> is too small.

## Miscellaneous

<code>-@ FILE</code>	Provide an extended command-line in file "FILE"
<code>-C, --cscore</code>	Use Cscore processing of the scorefile.
<code>--default-paths</code>	Reenables adding of directory of <code>CSD/ORC/SCO</code> to search paths, if it has been disabled by a previous <code>--no-default-paths</code> (e.g. in <i>.csoundrc</i> ).
<code>-D, --defer-gen1</code>	Defer GEN01 soundfile loads until performance time.
<code>--env:NAME=VALUE</code>	Set environment variable <code>NAME</code> to <code>VALUE</code> . Note: not all environment variables can be set this way, because some are read before parsing the command line. <code>INCDIR</code> , <code>SADIR</code> , <code>SFDIR</code> , and <code>SSDIR</code> are known to work.
<code>--env:NAME+=VALUE</code>	Append <code>VALUE</code> to ';' separated list of search paths in environment variable <code>NAME</code> (should be <code>INCDIR</code> , <code>SADIR</code> , <code>SFDIR</code> , or <code>SSDIR</code> ). If a file is found in multiple directories, the last will be used.
<code>--expression-opt</code>	<i>Note that this option has no affect in csound6. In Csound 5 only.</i> Turns on some optimizations in expressions: <ul style="list-style-type: none"> <li>• Redundant assignment operations are eliminated whenever possible. This means that for example this line <code>a1 = a2 + a3</code> will compile as <code>a1 Add a2, a3</code> instead of <code>#a0 Add a2, a3 a1 = #a0</code> saving a</li> </ul>

temporary variable and an opcode call. Less opcode calls result in reduced CPU usage (an average orchestra may compile about 10% faster with `--expression-opt`, but it depends largely on how many expressions are used, what the control rate is (see also below), etc.; thus, the difference may be less, but also much more).

- number of a- and k-rate temporary variables is significantly reduced. This expression

```
(a1 + a2 + a3 + a4)
```

will compile as

```
#a0 Add a1, a2
#a0 Add #a0, a3
#a0 Add #a0, a4      ; (the result is in #a0)
```

instead of

```
#a0 Add a1, a2
#a1 Add #a0, a3
#a2 Add #a1, a4      ; (the result is in #a2)
```

The advantages of less temporary variables are:

- less cache memory is used, which may improve performance of orchestras with many a-rate expressions and a low control rate (e.g. `ksmps = 100`)
- large orchestras may load faster due to less different identifier names
- index overflow errors (i.e. when messages like this Case2: `indx=-56004 (ffff253c); (short)indx = 9532 (253c)` are printed and odd behavior or a Csound crash occurs) may be fixed, because such errors are triggered by too many different (especially a-rate) variable names in a single instrument.

Note that this optimization (due to technical reasons) is not performed on i-rate temporary variables.



## Warning

When `--expression-opt` is turned on, it is not allowed to use the `i()` function with an expression argument, and relying on the value of k-rate expressions at i-time is unsafe.

<code>--version</code>	Exits after printing version information.
<code>--get-system-sr</code>	prints system sr and exits, requires previous <code>-o dac</code> . If the audio does not support this request then <code>-1</code> is reported.
<code>--help</code>	Display on-line help message.
<code>--devices</code>	Gives a list of available audio devices and then exits.

-I, --i-only	<i>i-time only</i> . Allocate and initialize all instruments as per the score, but skip all p-time processing (no k-signals or a-signals, and thus no amplitudes and no sound). Provides a fast validity check of the score pfields and orchestra i-variables. This option is exclusive of the --syntax-check-only flag.
--ignore_csopts=integer	If set to 1, Csound will ignore all options specified in the csd file's CsOptions section. See <i>Unified File Format for Orchestras and Scores</i> .
--ksmps=N	Set ksmps override to N (6.05 and later).
--max_str_len=integer	(min: 10, max: 10000) Maximum length of string variables + 1; defaults to 256 allowing a length of 255 characters. The length of string constants is not limited by this parameter.
-N, --notify	Notify (ring the bell) when score or MIDI track is done.
--no-default-paths	Disables adding of directory of CSD/ORC/SCO to search paths.
--no-expression-opt	Disables expression optimization.
-O FILE, --logfile=FILE	Log output to file <i>FILE</i> . If <i>FILE</i> is null (i.e. <i>-O null</i> or <i>--logfile=null</i> ) all printing of messages to the console is disabled.



### Note

This option only works from the command line and not from .csound6rc or a CsOptions section of a csd file. It needs to be acted upon before sound proper starts. It is ignored in .csoundrc and CsOptions.

--opcode-lib=LIBNAME	Load plugin library <i>LIBNAME</i> .
--orc orcname	Set the argument as the orchestra file. Used when not score is required>. (Csound 5.18 and later).
--syntax-check-only	Causes Csound to exit immediately after the orchestra and score parsers finish checking the syntax of the input files and before the orchestra performs the score. This option is exclusive of the --i-only flag. (Csound 5.08 and later).
-t0, --keep-sorted-score	Prevents Csound from deleting the sorted score file, score.srt, upon exit.
-U UTILITY, --utility=UTILITY	Invoke the utility program <i>UTILITY</i> . Use any invalid name to list the available utilities.
-x FILE, --extract-score=FILE	Extract a portion of the sorted score, score.srt, using the extract file <i>FILE</i> (see <i>Extract</i> ).

## Csound Environment Variables

The following environment variables can be used by Csound:

- SFDIR: Default directory for sound files. Used if no full path is given for sound files.

- **SSDIR**: Default directory for input (source) audio and MIDI files. Used if no full path is given for sound files. May be used in conjunction with **SFDIR** to set separate input and output directories. Please note that MIDI files as well as audio files are also sought inside **SSDIR**.
- **SADIR**: Default directory for analysis files. Used if no full path is given for analysis files.
- **SFOUTYP**: Sets the default output file type. Currently only 'WAV', 'AIFF' and 'IRCAM' are valid. This flag is checked by the **csound** executable and the utilities and is used if no file output type is specified.
- **INCDIR**: Include directory. Specifies the location of files used by *#include* statements.
- **OPCODE6DIR**: Defines the location of **csound** opcode plugins for the single precision float (32-bit) version.
- **OPCODE6DIR64**: Defines the location of **csound** opcode plugins for the double precision float (64-bit) version.
- **SNAPDIR**: Is used by the **FLTK** widget opcodes when loading and saving snapshots.
- **CSOUNDRC**: Defines the **csound** resource (or configuration) file. A full path and filename containing **csound** flags must be specified. This variable defaults to `.csoundrc` if not present.
- **CSSTRNGS**: In **Csound** 5.00 and later versions, the localisation of messages is controlled by two environment variables **CSSTRNGS** and **CS\_LANG**, both of which are optional. **CSSTRNGS** points to a directory containing `.xmg` files.
- **CS\_LANG**: Selects a language for **csound** messages.
- **RAWWAVE\_PATH**: Is used by the **STK** opcodes to find the raw wave files. Only relevant if you are using **STK** wrapper opcodes like **STKBowed** or **STKBrass**.
- **CSNOSTOP**: If this environment variable is set to "yes", then any graph displays are closed automatically at the end of performance (meaning that you possibly will not see much of them in the case of a short non-realtime render). Otherwise, you need to click "Quit" in the **FLTK** display window to exit, allowing for viewing the graphs even after the end of score is reached.
- **MFDIR**: Default directory for MIDI files. Used if no full path is given for MIDI files. Please note that MIDI files are sought in **SSDIR** and **SFDIR** as well.
- **CS\_OMIT\_LIBS**: Allows defining a list of plugin libraries that should be skipped. Libraries can be separated with commas, and don't require the "lib" prefix.

For more information about **SFDIR**, **SSDIR**, **SADIR**, **MFDIR** and **INCDIR** see *Directories and files*.

The only mandatory environment variables are **OPCODE6DIR** and **OPCODE6DIR64**. It is very important to set them correctly, otherwise most of the opcodes will not be available. Make sure you set the path correctly depending on the precision of your binary. If you run **csound** on a command line without any arguments you should see some text like: `Csound version 6.03.1 (double samples) May 10 2014`. This text refers to the double precision version.

**CSSTRNGS** and **CS\_LANG** currently have very limited use since **Csound** has not yet been completely translated into other languages.

Other environment variables which are not exclusive to **Csound** but which might be of importance are:

- **PATH**: The directory containing **csound** executables should be listed in this variable.

- *PYTHONPATH*: If you intend to use CsoundVST and python, the directory containing the *\_CsoundVST* shared library and the *CsoundVST.py* file must be in your *PYTHONPATH* environment variable (or the default path python searches in), so that the Python runtime knows how to load these files.
- *LADSPA\_PATH* and *DSSI\_PATH*: These environment variables are required if you are using the *dssi4cs* (LADSPA and DSSI host) plug-in opcodes.
- *CSDOCDIR*: Specifies the directory where the html help files are located. Though not used by Csound directly, this environment variable can help front-ends and editors (which implement it) to find the csound manual.

## Setting environment variables

### On the command line

You can set environment variables on the command line or the configuration file *.csoundrc* by using the command line flag `--env:NAME=VALUE` or `--env:NAME+=VALUE`, where *NAME* is the environment variable name, and *VALUE* is its value. See *Command-line Flags*



#### Note

Please note that this method of setting environment variables will not work for variables which are parsed before the command line arguments. *SADIR*, *SSDIR*, *SFDIR*, *INCDIR*, *SNAPDIR*, *RAWWAVE\_PATH*, *CSNOSTOP*, *SFOUTYP* should work, but the following environment variables must be set on the system prior to running *csound*: *OPCODE6DIR*, *OPCODE6DIR64*, *CSSTRINGS*, and *CS\_LANG*. *CSOUNDRC* can currently (v. 5.02) be set using `--env`, but this behavior is not guaranteed for future versions.

### Windows

To set a *csound* environment on Windows XP and 2000 go to Control Panel->System->Advanced and click on the button 'Environment Variables'. On other versions of Windows earlier than Windows XP and Windows 2000 you set environment variables in the *autoexec.bat* file. Go to 'My Computer', select C: drive, right click on *autoexec.bat*, and select 'Edit'. The statement format is: `SET NAME=VALUE` .

### Linux

You can set environment variables on Linux in many ways. You can set them using the *export* shell command, by setting them on *.bashrc* or similar files or by adding them to the */etc/profile* file.

### Mac

```
~$ export OPCODE6DIR64=/Users/you/your/Csound6/build
```

in addition if the bash shell is the default, then it is usually easier to edit your *.bashrc* or */etc/profile*.

Note that if users choose one of the above methods, ie editing the *.bashrc* file then the environment variables are executed when a new shell is created. This can be problematic if your application implements a Quartz or Aqua interface and does not use the commandline.

If this is the case, then the standard solution (up to OS 10.3.9 and unless the application uses the *csoundAPI* and sets the environ variables directly) is to create an XML property list file (called a *.plist* file by the OS). This file should nominally be located at *~/MacOSX/Environment.plist*. This has been a solution specifically for the [*csoundapi~*] object for Pd on OS X. Since Pd uses an OS X native *.app* style packaging,

and runs off of the Aqua interface, the standard means of supplying environment variables to Csound do not work. The solution is to set Csound's environment variables for the Aqua environment.

Likely, most users will not have the hidden folder `.MacOSX` located in their `$HOME` directory (aka `~/`) This folder must first be created and the `Environment.plist` added to this folder. The contents of the `Environment.plist` file should be something like:

```
<?xml version="1.0" encoding='UTF-8'?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
    "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
<key>OPCODEDIR</key>
<string>/Library/Frameworks/CsoundLib.framework/Versions/5.1/Resources/Opcodes</string>
<key>OPCODEDIR64</key>
<string>/Volumes/ExternalHD/devel/csound5/lib64</string>
<key>INCDIR</key>
<string>/Volumes/ExternalHD/CSOUND/include</string>
<key>SFDIR</key>
<string>/Volumes/ExternalHD/iTunes/csoundaudio</string>
</dict>
</plist>
```

and so on, using the XML `<key>` tag for each environment variable required by the API and the `<string>` tag for it's corresponding path on the system.

Please note that you must login out and login in for these changes to take effect.

## Unified File Format for Orchestras and Scores

### Description

The Unified File Format, introduced in Csound version 3.50, enables the orchestra and score files, as well as command line flags, to be combined in one file. The file has the extension `.csd`. This format was originally introduced by Michael Gogins in AXCSound.

The file is a structured data file which uses markup language, similar to any SGML such as HTML. Start tags (`<tag>`) and end tags (`</tag>`) are used to delimit the various elements. The file is saved as a text file.

### Structured Data File Format

#### Mandatory Elements

The first tag in the file must be the start tag `<CsoundSynthesizer>`. The last tag in the file must be the end tag `</CsoundSynthesizer>`. This element is used to alert the csound compiler to the `.csd` format. All text before the start tag and after the end tag is ignored by Csound. The tag may also be spelled `<CsoundSynthesiser>`.

#### Options (`<CsOptions>`)

Csound *command line flags* are put in the Options Element. This section is delimited by the start tag `<CsOptions>` and the end tag `</CsOptions>` Lines beginning with `#` or `;` are treated as comments.

#### Orchestra (`<Csinstruments>`)

The instrument definitions (orchestra) are put into the Instruments Element. The statements and syntax in this section are identical to the Csound *orchestra file*, and have the same requirements, including the

header statements (*sr*, *kr*, etc.) This Instruments Element is delimited with the start tag `<CsInstruments>` and the end tag `</CsInstruments>`.

## Score (`<CsScore>`)

Csound score statements are put in the Score Element. The statements and syntax in this section are identical to the Csound *score file*, and have the same requirements. The Score Element is delimited by the start tag `<CsScore>` and the end tag `</CsScore>`.

As an alternative Csound score statements can also be generated by an external program using the CsScore scheme with an attribute *bin*. The lines upto the closing tag `</CsScore>` are copied to a file and the external program named is called with that file name and the destination score file. The external program should create a standard Csound score.

## Optional Elements

### Included Base64 Files (`<CsFileB>`)

Base64-encoded files may be included with the tag `<CsFileB filename=filename>`, where *filename* is the name of the file to be included. The Base64-encoded data should be terminated with a `</CsFileB>` tag. For encoding files, the *csb64enc* and *makecsd* utilities (included with Csound 5.00 and newer) can be used. The file will be extracted to the current directory, and deleted at end of performance. If there is an already existing file with the same name, it is not overwritten, but an error will occur instead.

Base64-encoded MIDI files may be included with the tag `<CsMidifileB filename=filename>`, where *filename* is the name of the file containing the MIDI information. There is no matching end tag. This was added in Csound version 4.07. Note: using this tag is not recommended; use `<CsFileB>` instead.

Base64-encoded sample files may be included with the tag `<CsSampleB filename=filename>`, where *filename* is the name of the file containing the sample. There is no matching end tag. This was added in Csound version 4.07. Note: using this tag is not recommended; use `<CsFileB>` instead.

### Included Unencoded Files (`<CsFile>`)

Unencoded files may be included with the tag `<CsFile filename=filename>`, where *filename* is the name of the file to be included. The data should be terminated with a `</CsFile>` tag alone on a line. The file will be extracted to the current directory, and deleted at end of performance. If there is an already existing file with the same name, it is not overwritten, but an error will occur instead.

### Version Blocking (`<CsVersion>`)

Versions of Csound may be blocked by placing one of the following statements between the start tag `<CsVersion>` and the end tag `</CsVersion>`:

Before `##`

or

After `##`

where `##` is the requested Csound version number. The second statement may be written simply as:

`##`

This was added in Csound version 4.09.

## Licence Information (<CsLicence> or <CsLicense>)

Licensing details can be included in between the start tag `<CsLicence>` and the end tag `</CsLicence>`. There is no format for this information, any text is acceptable. This text will be printed by Csound to the console when the CSD is run.

## Licence Information (<CsShortLicence> or <CsShortLicense>)

From version 6.05 licensing details can be also included in between the start tag `<CsShortLicence>` and the end tag `</CsShortLicence>`. This offers seven well-known licences, coded as an integer.

- 0: "All rights reserved" (default)
- 1: "Creative Commons Attribution-NonCommercial-NoDerivatives (CC BY-NC-ND)"
- 2: "Creative Commons Attribution-NonCommercial-ShareAlike (CC BY-NC-SA)"
- 3: "Creative Commons Attribution-NonCommercial (CC BY-NC)"
- 4: "Creative Commons Attribution-NoDerivatives (CC BY-ND)"
- 5: "Creative Commons Attribution-ShareAlike (CC BY-SA)"
- 6: "Creative Commons Attribution-ShareAlike (CC BY)"
- 7: "Licenced under BSD"

## Embedded HTML (<html>)

Any valid HTML code can be embedded in the CSD file. This code should be structured exactly like an ordinary Web page. This code can contain any valid HTML, JavaScript, Cascading Style Sheet, WebGL, etc., etc. code.

In some Csound front ends and programming environments, including at least CsoundQt or Csound for Android, this page will be parsed, executed, and displayed by a Web browser embedded in the environment. JavaScript code in this page will have access to a global `csound` object that implements the following functions, which are a selected subset of the Csound API. The names, data types, and uses of these functions are exactly the same as detailed in the Csound API Reference Manual.

```
[int] getVersion ();
compileOrc (orchestra_text);
[double] evalCode (orchestra_expression);
readScore (score_text);
setControlChannel (channel_name, numeric_value);
[double] getControlChannel (channel_name);
message (message_string);
[int] getSr ();
[int] getKsmps ();
[int] getNchnls ();
// Not a part of the Csound API -- called by the environment to detect whether Csound is running.
[int] isPlaying ();
```

The HTML element of the CSD file can be used to create custom user interfaces for the piece, to generate score events and even orchestra code using JavaScript, to store presets for widgets, and for many other purposes. The *GameOfLife3D.csd* [examples/GameOfLife3D.csd] and *LindenmayerCanvas.csd* [examples/LindenmayerCanvas.csd] examples demonstrate these uses (tested in CsoundQt; running these examples requires additional resources found in the Csound examples directory in GIT).

## Example

Below is a sample file, `test.csd`, which renders a `.wav` file at 44.1 kHz sample rate containing one second of a 1 kHz sine wave. Displays are suppressed. `test.csd` was created from two files, `tone.orc` and `tone.sco`, with the addition of command line flags.

```

<CsoundSynthesizer>
  ; test.csd - a Csound structured data file

<CsOptions>
  -W -d -o tone.wav
</CsOptions>

<CsVersion>      ; optional section
  Before 4.10    ; these two statements check for
  After 4.08     ; Csound version 4.09
</CsVersion>

<CsInstruments>
  ; originally tone.orc
  sr = 44100
  kr = 4410
  ksmpls = 10
  nchnls = 1
  instr 1
    al oscil p4, p5, 1 ; simple oscillator
    out al
  endin
</CsInstruments>

<CsScore>
  ; originally tone.sco
  f1 0 8192 10 1
  i1 0 1 20000 1000 ; play one second of one kHz tone
  e
</CsScore>

</CsoundSynthesizer>

```

## Command Line Parameter File (.csoundrc)

If the file *.csoundrc* exists, it will be used to set the command line parameters. These can be overridden. Csound 5.00 and newer versions read this file from the HOME directory first (or the full path file name defined by the CSOUNDRC *environment variable*), and then the current directory. If both exist, options in the *.csoundrc* in the current directory will have higher precedence. It uses the same form as a *.csd* file, but no tags are needed. Lines beginning with # or ; are treated as comments.

A *.csoundrc* file can contain something like this:

```
-+rtaudio=portaudio -odac2 -iadc2 -+rtmidi=winmme -M1 -Q1 -m0
```

In this case, csound will generate real-time output and take realtime input from device 2, using the portaudio driver interface. It will input and output realtime MIDI on interface 1. It will print very few messages (-m0). These options will be used by default when other options are not given inside the <CsOptions> of the *.csd* file or the command line (See *Order of precedence*).

## Score File Preprocessing

### The Extract Feature

This feature will extract a segment of a sorted numeric score file according to instructions taken from a control file. The control file contains an instrument list and two time points, from and to, in the form:

```
instruments 1 2 from 1:27.5 to 2:2
```

The component labels may be abbreviated as i, f and t. The time points denote the beginning and end of the extract in terms of:

```
[section no.] : [beat no.].
```

Each of the three parts of the argument is optional. The default values for missing i, f or t are:

```
all instruments, beginning of score, end of score.
```

## Independent Pre-Processing with Scsort

Although the result of all score preprocessing is retained in the file `score.srt` after orchestra performance (it exists as soon as score preprocessing has completed), the user may sometimes want to run these phases independently. The command

```
scot filename
```

will process the Scot formatted filename, and leave a *standard numeric score* result in a file named `score` for perusal or later processing.

The command

```
scsort < infile > outfile
```

will put a numeric score `infile` through Carry, Tempo, and Sort preprocessing, leaving the result in `outfile`.

Likewise *extract*, also normally invoked as part of the *Csound command*, can be invoked as a standalone program:

```
extract xfile < score.sort > score.extract
```

This command expects an already sorted score. An unsorted score should first be sent through Scsort then piped to the extract program:

```
scsort < scorefile | extract xfile > score.extract
```

---

# Using Csound

Csound can be operated in a variety of modes and configurations. The original method for running Csound was as a console program (DOS prompt for Windows, Terminal for Mac OS X). This, of course, still works. Running `csound` without any arguments prints out a list of command-line options, which are more fully explained in the *Command Line Flags (by Category)* section. Normally, the user executes something like:

```
csound myfile.csd
```

or separate orchestra (`orc`) and score (`sco`) files can be used:

```
csound myorchestra.orc myscore.sco
```

You can find many `.csd` files in the examples folder. Most opcode entries in this manual also include simple `.csd` files showing the usage of the opcode.

There are also many *Front-Ends* which can be used to run `csound`. A *Front-End* is a graphical program that eases the process of running `csound`, and sometimes provides editing and composing functions.

Csound also has several ways of producing output. It can:

- Read and write soundfiles (off-line rendering) - Using the `-o` and `-i` flags specifying an output filename.
- Read and write digital audio using a sound card (real-time rendering) - Using the `-odac` and `-iadc` flags
- Read and write MIDI files (non-realtime) - Using the `-F` and `--midioutfile` flags.
- Read and write MIDI using a MIDI interface and controller (real-time control) - Using the `-M` and `-Q` flags.

## Csound's Console Output

When Csound runs, it prints a text output to the console, which shows data about the Csound run. A Console output looks something like this:

```
time resolution is 0.455 ns
PortMIDI real time MIDI plugin for Csound
virtual_keyboard real time MIDI plugin for Csound
PortAudio real-time audio module for Csound
0dBFS level = 32768.0
Csound version 5.10 beta (float samples) Apr 19 2009
libsndfile-1.0.17
Reading options from $HOME/.csoundrc
UnifiedCSD:  oscil.csd
STARTING FILE
Creating options
Creating orchestra
Creating score
orchname:  /tmp/csound-XYACV6.orc
scorename: /tmp/csound-IYtLAJ.sco
rtaudio:  ALSA module enabled
rtmidi:   PortMIDI module enabled
orch compiler:
17 lines read
      instr  1
Elapsed time at end of orchestra compile: real: 0.129s, CPU: 0.020s
sorting score ...
      ... done
Elapsed time at end of score sort: real: 0.130s, CPU: 0.020s
```

```
Csound version 5.10 beta (float samples) Apr 19 2009
displays suppressed
0dBFS level = 32768.0
orch now loaded
audio buffered in 256 sample-frame blocks
ALSA input: total buffer size: 1024, period size: 256
reading 1024-byte blks of shorts from adc (RAW)
ALSA output: total buffer size: 1024, period size: 256
writing 1024-byte blks of shorts to dac
SECTION 1:
ftable 1:
new alloc for instr 1:
B 0.000 .. 2.000 T 2.000 TT 2.000 M: 10000.0 10000.0
Score finished in csoundPerform().
inactive allocs returned to freespace
end of score.                overall amps: 10000.0 10000.0
                           overall samples out of range: 0 0
0 errors in performance
Elapsed time at end of performance: real: 2.341s, CPU: 0.050s
345 1024-byte soundblks of shorts written to dac
Removing temporary file /tmp/csound-CoVcrm.srt ...
Removing temporary file /tmp/csound-IYtLAJ.sco ...
Removing temporary file /tmp/csound-XYACV6.orc ...
```

The console output of Csound is quite verbose, particularly before the actual performance (like version, plugins loaded, etc.). Performance actually started when the console printed:

```
SECTION 1:
```

In this particular run, the lines:

```
new alloc for instr 1:
B 0.000 .. 2.000 T 2.000 TT 2.000 M: 10000.0 10000.0
```

Show that a single note for instrument 1, that lasted 2 seconds starting at time 0.000, was produced with an amplitude of 10000 for both channel 1 and 2. An important section of the console output is:

```
end of score.                overall amps: 10000.0 10000.0
                           overall samples out of range: 0 0
```

Which shows the overall amplitude and the number of samples which were clipped because they were out of range.

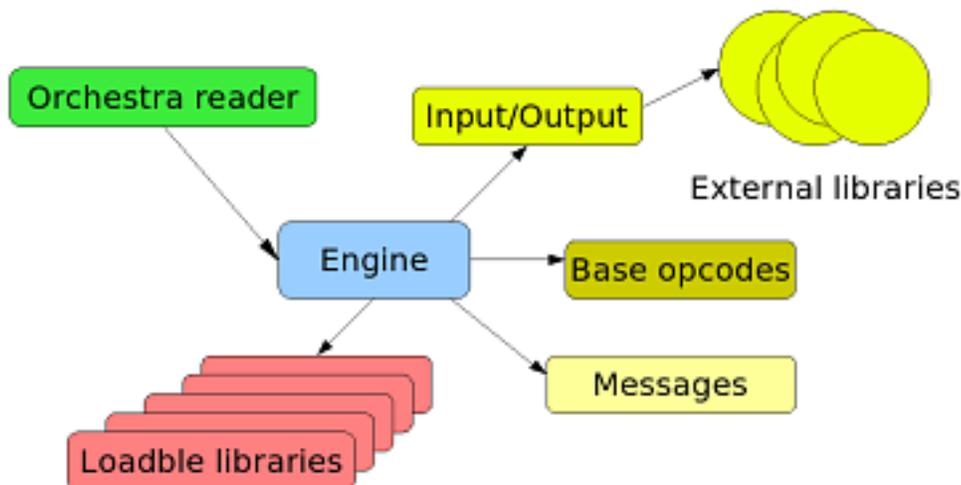
The line:

```
Elapsed time at end of performance: real: 2.341s, CPU: 0.050s
```

Shows the clock time and the CPU time it takes for the processor to complete the task. If CPU time is lower than clock time it means the csd can run in realtime (unless it has some sections which are extremely CPU intensive). The "real time" figure is the total running time and it is larger because it accounts for disk access, module loading, etc. (CPU time is strictly number-crunching time). If you have a sound that lasts for 100s and it takes 5s to generate it offline, it means that you are taking around 5% of CPU, and that it runs on 0.05 of realtime.

## How Csound works

Csound processes and generates output using "unit generators" (ugens) called *opcodes*. These opcodes are used to define *instruments* in the *orchestra*. When you run Csound, the engine loads the base Opcodes, and the opcodes contained in separate loadable "opcode libraries". It then interprets the orchestra (through the orchestra reader). The engine sets up an instrument processing chain, which then receives events from the score or in real-time. The processing chain uses the input/output modules to generate output. There are modules that can write to file, or generate *real-time audio output*.



The Csound Modular structure.

## Csound's processing buffers

Csound processes audio in sample blocks called buffers. There are three separate buffer layers:

1. *spout* = Csound's innermost software buffer, contains *ksmps* sample frames. Csound processes real-time control events once every *ksmps* sample frames.
2. *-b* = Csound's intermediate software buffer (the "software" buffer), in sample frames. Should be (but does not need to be) an integral multiple of *ksmps* (can equal *ksmps* too). Once per *ksmps* sample frames, Csound copies *spout* to the *-b* buffer. Once per *-b* sample frames, Csound copies the *-b* buffer to the *-B* "hardware" buffer.
3. *-B* = The sound card's internal buffer (the "hardware" buffer), in sample frames. Should be (and may need to be) an integral multiple of *-b*. If Csound misses delivering a *-b* one time, the extra *-b* sample frames in *-b* are still there for the sound card to keep playing while Csound catches up. But they can be the same size if you're willing to bet Csound can always keep up with the sound card.

## Amplitude values in Csound

Amplitude values in Csound are always relative to a "*Odbfs*" value representing the peak available amplitude before clipping, in either an AD/DA codec, or in a soundfile with a defined range (which both WAVE and AIFF are). In the original Csound, this value was always 32767, corresponding to the bipolar range of a 16bit soundfile or 16bit AD/DA codec, Csound's only possible output back then. This remains the *default* peak amplitude for Csound, for backward compatibility and you will find some of this manual's examples still use this value (hence you find large amplitude values like 10000).

The *Odbfs* value enables Csound to produce appropriately scaled values to whatever output format is being used, whether 24bit integer, 32bit floats, or even 32bit integers. Put another way, the literal amplitude values you write in a Csound instrument only match those written *literally* to the file if the *Odbfs* value

in Csound corresponds exactly to that of the output sample format. The consequence of this approach is that you can write a piece with a certain amplitude and have it render correctly and identically (setting aside of course the better dynamic range of the high-res formats) whether written to an integer or floats file, or rendered in real-time.



## Note

The one exception to this is if you choose to write to a "raw" (headerless) file format. In such cases the internal *0dbfs* value is meaningless, and whatever values you use are written unmodified. This does enable arbitrary data to be generated or processed by Csound. It is a relatively exotic thing to do, but some users need it.

You can choose to redefine the *0dbfs* value in the orchestra header, purely for your own convenience or preference. Many people will choose 1.0 (the standard for SAOL, other software like Pure Data, and for many plugin standards such as VST, LADSPA, CoreAudio AudioUnits, etc), but any value is possible.

The common factor in defining amplitudes is the decibel (dB) scale, with  $0\text{dB}_{\text{FS}}$  always understood as digital peak; hence "0dbfs" means "0dB Full-Scale value". This measure is different to actual amplitude values, since amplitude values are a linear scale which show the actual oscillation around 0, so they can be positive or negative. Decibel values are an absolute logarithmic scale, but can be useful for most opcodes as well. You can convert amplitude to and from decibels using the *ampdb*, *ampdbfs*, *dbamp* and *dbfsamp* functions. This way, Csound enables the programmer to express all amplitudes in dB - lower amplitudes will then be represented by negative dB values. This reflects industry practice (e.g. in level meters in mixers, etc).

For example the same dB level of -6dB (half the amplitude) or -20dB are actually a different linear amplitude according to 0dbfs like this:

**Table 2.  $\text{dB}_{\text{FS}}$  in relation to amplitude**

$\text{dB}_{\text{FS}}$	0dbfs = 32767 (default)	0dbfs = 1	0dbfs = 1000 (unusual)
0 dB	32767	1	1000
-6 dB	16384	0.5	500
-20 dB	3276.7	0.1	100

Some Csound users might therefore be minded to express all levels in  $\text{dB}_{\text{FS}}$ , and obviate any confusion or ambiguity of level that may otherwise arise when using explicit amplitude values. The decibel scale reflects the response of the ear pretty closely, and that when you want to express a really quiet level, it might be easier and more expressive to write "-46dB" than "0.005" or "163.8".

The reason for using 0dbfs is very simple: digital peak equates to maximum level regardless of sample resolution. If you then define a signal at -110dB you will lose it if rendering to a 16bit file, but retain it (audibly or not) if rendering to 24bit or better. In other words, there is a fixed ceiling, but a moveable floor - you can define sounds as quietly as you like (e.g. envelope tails), in a predictable way, and preserve them or not (without changing the orch code at all), depending on the resolution (file or audio i/o) you render to.



## A note on digital amplitude, decibels and dynamic range

A convenient approximation of dynamic range for a certain digital precision is to calculate the decibel interval between the minimum value and the maximum value for a sample. As a rule of thumb, 1 bit (doubling of level) is 6dB, so 16bits = 96dB.

This is not entirely accurate since audio sample values are represented on a bipolar scale with positive and negative values, and 1 bit is used for the sign. Therefore, for 16bit integer

samples actually use 1 bit for the sign and 15 bits for the values, so the actual dynamic range is 90dB.

## Real-Time Audio

The following information applies mostly to `csound` being run directly from the command line. Front-ends implement these features in different ways, but knowledge of them is necessary in some of them.

The `-i` and `-o` flags can be used to specify realtime output instead of the ordinary non-realtime file output. You should use `-o dac` for realtime output and `-i adc` for realtime input. Naturally, you can use either one or both if your hardware supports it. You can also specify the hardware you want to use by appending a device number or name to the flag (See `-i` and `-o`).

You might also need to use the `-+rtaudio` flag to specify the driver interface to be used. Csound defaults to using Portaudio, which is cross-platform and reliable, but for better performance, you might need to use ALSA or JACK on linux, and CoreAudio on Mac. You can use ASIO on Windows if your version of Portaudio has been compiled with ASIO support.

You can see a list of available devices by giving a device number which is out of range, for instance `-o dac99`. This will also reveal if you have ASIO available if you are using PortAudio.

## Period & Buffer Sizes

Period and buffer sizes will vary greatly from one machine to another. Lower buffer sizes will result in lower latency, but might cause breakups or clicks in the audio. The Csound flags which control period and buffer sizes are `-b` and `-B`, respectively. Buffer size is hardware dependant, and some experimentation may be necessary to find the optimal balance between low latency performance and uninterrupted audio output. The values given to `-b` and `-B` should be powers of two, and the value of `-B` should be at least one power of two higher than that of `-b`.

Currently, with `-B` set to 512, audio output latency is about 12 milliseconds, fast enough for reasonably responsive keyboard playing. Even shorter latencies, are feasible on some systems.

## Control Rate

Low values for `ksmps` will in general give a higher quality of synthesis, but will consume more system resources. There is no hard and fast rule for setting `ksmps` - different orchestras will require different control rates. A waveguide instrument will need a `ksmps` of 1 (and may not be suitable for realtime use), whereas a simple FM synth may be run with a higher `ksmps` without noticeable degradation of sound. If the FM synth were to be used to play a monophonic bassline, a very low `ksmps` may be used, however more complex note clusters will require a higher `ksmps`. A well-tuned Linux system should be capable of running even complex polyphonic synths with `ksmps` values as low as 4 or 8. If full duplex audio is required, `-b` must be an integer multiple of `ksmps`. Bearing this in mind, a rule of thumb might be to only use powers of two for `ksmps`.

Some settings differ according to platform. See further below for information for each platform.

## Realtime I/O on Linux

Under Linux, the default portaudio/portmidi settings will result in higher latency than that which can be achieved using ALSA and/or JACK (see a separate manual section on this). The portaudio/portmidi plugins are audio and MIDI servers, which provide an interface to the ALSA drivers, in a manner which is in some respects similar but fundamentally different from that provided by JACK.

## Using ALSA

The highest level of control and the lowest possible level of latency are to be achieved using the ALSA plugins in combination with the `--sched` flag. Using `--sched` requires that Csound be run as the root user, which may be impossible or undesirable in some circumstances.

The ALSA plugins require the "name" of a "card" and a "device". Unless you have named your "cards" in `~/asoundrc` (or `/etc/asound.conf`), the "names" will actually be numbers. In order to obtain a list of the possible configurations, use the command line utilities "aplay", "arecord" and "amidi". These utilities are included with most Linux distros, or can be downloaded and built from source here:

<ftp://ftp.alsa-project.org/pub/utils/>



### Note

On every boot, the soundcard may have a different hardware order number, especially when there are more soundcards in the system. This can be awkward as every time you have to set the right number again. You can assign a fixed order by adding some lines to `/etc/modprobe.d/alsa-base-conf`, for example for a card with the ice1712 chip :

```
# ALSA module ordering for soundcard
options snd slots=snd_ice1712
```

## Audio Output

Running the following command:

```
aplay -l
```

will give you a list of the audio playback devices available on your system. Typically this list will look something like:

```
[...]
**** List of PLAYBACK Hardware Devices ****
card 0: A5451 [ALI 5451], device 0: ALI 5451 [ALI 5451]
[...]
```

If you have more than one card on your system, or if there is more than one device on your card, the list will of course be more complicated, however in all cases the information that is pertinent is the number/name of the card/device. In order to use the above soundcard for audio output, the following flag would be added to the Csound command line, `~/csoundrc`, or the `<CsOptions>` section of a CSD:

```
--rtaudio=alsa -o dac
```

## Output with dmix

If you would like to use Csound with dmix and your soundcard does not support hardware mixing of audio streams, special care is needed in setting up of software (-b) and hardware (-B) buffers. If you get a message from Csound's ALSA driver that looks like the following:

ALSA: -B 8192 not allowed on this device; use 7526 instead

there is a good chance that you may be using dmix. If you are using dmix, the `-b` and `-B` settings of Csound must be synced the `period_size` and `buffer_size` of dmix respectively, using a ratio of the `sr` for the Csound project to the sample rate that dmix is set up to. The following formula will determine what settings to use for Csound given the settings of dmix:

```
-b = (csound_sr/dmix_sample_rate) * dmix_period_size
-B = (csound_sr/dmix_sample_rate) * dmix_buffer_size
```

For example, if dmix is set to 48000 sample rate, a `period_size` of 1024, and a `buffer_size` of 8192, when running a Csound project with `sr=48000`, the settings for buffers should be `"-b 1024 -B8192"`. If the `sr=24000`, the settings for buffers should be `"-b 512 -B4096"`.

Because of this relationship, if a Csound project's `sr` does not evenly divide into the `sample_rate` used by dmix, then it may be difficult if not impossible to set the correct setting for `-b` and `-B` due to rounding errors. It is suggested then that if you are using sample rates different than what your setting is for dmix, then you may want to configure dmix to have a `period_size` and `buffer_size` that can be evenly divided by the ratio between the `csound sr` and `dmix sample_rate`. For example, to run a project with `sr=16000`, the following dmix setting:

```
pcm.amix {
    type dmix
    ipc_key 50557
    slave {
        pcm "hw:0,0"
        period_time 0
        #period_size 1024
        #buffer_size 8192
        period_size 1536
        buffer_size 12288
    }
    bindings {
        0 0
        1 1
    }
}

# route ALSA software through pcm.amix
pcm.!default {
    type plug
    slave.pcm "amix"
}
```

with `period_size 1536` and `buffer_size 12288` will divide nicely by 3 (the ratio of the `csound sr` to the `dmix sample_rate`) to get `"-b 512 -B4096"` ( $(16000/48000) * 1536 = 512$ ,  $(16000/48000) * 12288 = 4096$ ).



## Note

For most soundcards that this affects, the default sample rate for the card will be 48000 and the defaults for dmix will be 1024 and 8192.

## Audio Input

Typically the same card will be used for both input and output, so to continue using the foregoing example, the flag:

```
-i adc:hw:0,0
```

would be added for audio input from Card 0 Device 0. To use the default card employ one of the following flags, with the forementioned warning that this will not necessarily work:

```
-i adc
```

If you wish to use a different card or device for input, running the following utility from the command line will provide a list of input devices:

```
arecord -l
```

If, by way of an example, you wanted to use a USB audio interface, which is the second "card" in your system, for output, but wanted to use your internal soundcard, the first card in your setup, for input, you would put the following flags somewhere useful:

```
--rtaudio=alsa -i adc:hw:0,0 -o dac:hw:1,0
```

If you wanted to use the second device on your USB interface, to send audio to a specific channel, for instance, you would use the following flags:

```
--rtaudio=alsa -i adc:hw:0,0 -o dac:hw:1,1
```

## MIDI

2 Midi drivers are available:

- Raw Midi.
- Alsa Sequencer (since version 5.18).

## MIDI Input (Raw Midi driver)

In order to enable your orchestra to receive MIDI input you can use VirMIDI or MIDIThru, whichever you prefer. Setting up these virtual MIDI ports is a topic that has been covered extensively elsewhere, see The Linux MIDI how-to [<http://www.midi-howto.com/>] or browse your distro's documentation or the ALSA documentation for instructions on how to install and configure either VirMIDI or MIDIThru (seqdummy). Once you have done so run:

```
amidi -l
```

for a list of available devices. Typically this will look something like the following:

```
[...]  
Device Name  
hw:1,0 Virtual Raw MIDI (16 subdevices)  
hw:1,1 Virtual Raw MIDI (16 subdevices)  
hw:1,2 Virtual Raw MIDI (16 subdevices)  
hw:1,3 Virtual Raw MIDI (16 subdevices)  
hw:2,0,0 PCR MIDI  
hw:2,0,1 PCR 1
```

In this example, Csound can connect to any of the four available Virtual Raw MIDI ports, where it will listen for MIDI input. The following flag instructs Csound to listen on the first of these ports:

```
--rtmidi=alsa -Mhw:1,0
```

You will then need to connect your hardware or software controller to the port which is hosting your Csound synthesizer. The simplest way to do this is using the "aconnect" utility. Run:

```
aconnect -li
```

for a list of available input devices, and:

```
aconnect -lo
```

for a list of available output devices (including the port to which Csound has been connected). These should give something like the following:

```
#aconnect -li
client 0: 'System' [type=kernel]
  0 'Timer'
  1 'Announce'
    Connecting To: 15:0
client 20: 'Virtual Raw MIDI 1-0' [type=kernel]
  0 'VirMIDI 1-0'
client 21: 'Virtual Raw MIDI 1-1' [type=kernel]
  0 'VirMIDI 1-1'
client 22: 'Virtual Raw MIDI 1-2' [type=kernel]
  0 'VirMIDI 1-2'
client 23: 'Virtual Raw MIDI 1-3' [type=kernel]
  0 'VirMIDI 1-3'
client 24: 'PCR' [type=kernel]
  0 'PCR MIDI'
  1 'PCR 1'
  2 'PCR 2'
```

```
#aconnect -lo
client 20: 'Virtual Raw MIDI 1-0' [type=kernel]
  0 'VirMIDI 1-0'
client 21: 'Virtual Raw MIDI 1-1' [type=kernel]
  0 'VirMIDI 1-1'
client 22: 'Virtual Raw MIDI 1-2' [type=kernel]
  0 'VirMIDI 1-2'
client 23: 'Virtual Raw MIDI 1-3' [type=kernel]
  0 'VirMIDI 1-3'
client 24: 'PCR' [type=kernel]
  0 'PCR MIDI'
  1 'PCR 1'
```

In the following example, the USB keyboard which is listed above as client 24 will be connected to a Csound synthesizer which is listening on the first VirMIDI port. The keyboard has three output ports. The first (24:0) transmits messages received on the MIDI in port, the second (24:1) transmits keyboard and controller messages, and the third (24:2) transmits system exclusive messages. The following command connects the second port of the keyboard to the Csound synthesizer:

```
aconnect 24:1 20:0
```

Remember that Csound acts as a raw MIDI device and is not an ALSA sequencer client. This means that Csound will not appear in MIDI device listings and will not be available for use directly with *aconnect*, so you must connect to a virtual device (like 'virtual raw MIDI' or 'MIDI through') for persistent connections, or connect directly to the destination using command line flags.

## MIDI Output (Raw Midi driver)

Csound can be connected to any device which shows up on the ALSA sequencer list of output ports, obtained by "amidi -l" as above. In order to connect a Csound synthesizer to the MIDI out port of the keyboard listed above, the following flag would be used:

```
-Qhw:2,0,0
```

## MIDI Input and Output (Midi Sequencer driver)

This driver is to be preferred over the Raw Midi driver. It has these advantages:

- Multiple concurrent access.
- Scheduled by priority queues.
- Real-time event dispatching i.e., the role of the Midi Sequencer is to deliver events at the right time (sequence) to the right destination (device).

The following command will call the Midi Sequencer. Here it listens to midi port 20. The midi output port is also 20:

```
--rtmidi=alsaseq -M20 -Q20
```

Csound will automatically create its own ALSA sequencer port. For a list of available devices, use the following command:

```
aconnect -i -o
```

This will create output that will look something like the following:

```
client 0: 'System' [type=kernel]
  0 'Timer'
  1 'Announce'
client 14: 'Midi Through' [type=kernel]
  0 'Midi Through Port-0'
client 20: 'M Audio Delta 1010' [type=kernel]
  0 'M Audio Delta 1010 MIDI'
client 130: 'Csound' [type=user]
  0 'Csound'
```

The output of Csound will contain lines like:

```
.....
ALSASEQ: opened MIDI output sequencer
ALSASEQ: created output port 'Csound' 130:0
ALSASEQ: connected to 20:0
```

```

.....
.....
ALSASEQ: opened MIDI input sequencer
ALSASEQ: created input port 'Csound' 130:0
ALSASEQ: connected from 20:0
.....

```

## Scheduling

If you are able to run Csound as the root user, using the "--sched" flag will dramatically improve realtime performance, when using ALSA, however you may hang your system if you do something stupid. DO NOT use "--sched" if you are using JACK for audio output. JACK controls scheduling for the audio applications connected to it, and also tries to run at the highest possible priority. If the "--sched" flag is used, Csound and JACK will be competing rather than cooperating, resulting in extremely poor performance.

## Using Pulseaudio

Support for Pulseaudio [<http://www.pulseaudio.org/>] was added in Csound 5.09. You can specify the following settings:

1. Sink names: it's possible to use a number instead of the full name, so `-odac:1` would select your second device (count starts at 0).
2. Server name: it's possible to connect to a specific server by using `++server=<server_string>` where `server_string` is a name of a server or a more complex server selection string (see [pulseaudio.org](http://www.pulseaudio.org/) [<http://www.pulseaudio.org/>] on server strings). This should be network transparent and should allow connections to remote machines.
3. Stream names: it is possible to label the streams generated by csound, by using `++output_stream=<stream-name>` and `++input_stream=<stream-name>`

Here's an example command line:

```
csound -odac:1 examples/trapped.csd ++rtaudio=pulse ++server=unix:/tmp/pulse-victor/native ++output_stream=trapped
```

## Mac OSX

### Real-time Audio

OSX users can use either the *PortAudio* (default), *auhal* (or *coreaudio*), or the *Jack* realtime audio modules. The *auhal* module is a native OSX module which provides good latency, but it might not work with some external hardware. The *Jack* module can be used for interconnecting with other applications, but you will need to install the *JackOSX* software in order to use it. To activate a realtime module, you can use the `++rtaudio` flag with value of *portaudio*, *auhal*, or *jack*. The default value is *portaudio*, which is activated by default without specifying it.

You also need to specify the sound device you want to use, and specify that you want to generate real-time audio output instead of soundfile to disk output. To do this, you must use the `-odac` or `-o dac` flag, which tells csound to output to the Digital-to-Analog converters instead of a file. By adding a number after the flag (e.g. `-odac2`), you can choose the device number you want. To find out available devices in your system, you can use a large out of range number (e.g. `-odac99`), and Csound will report an error, and list

available devices. This numbering convention works for portaudio and auhal, but for Jack, you will need to pass the name of the desired output device after a colon (e.g. `-odac:system:playback_`).

Enabling realtime audio input is done using `-iadc`, which makes csound listen to the realtime audio inputs. You can again select the device by its number (or name), and check for available devices using an out of range number. Note that for input you use 'adc' instead of 'dac'. Make sure you have the appropriate input selected in your soundcard's control panel.

## Real-time MIDI

To enable Real-time MIDI on OSX, you can use the `-M` flag for MIDI input and the `-Q` flag for MIDI output. You might need to specify the device number after the flag (e.g. `-M2`), and again, you can find the available devices by giving an out of range number.

Csound will use PortMidi as the default MIDI module, but there's also a native coremidi module, which can be activated with the flag:

```
-+rtmidi=cmidi
```

The coremidi module currently only supports MIDI input.

A typical set of flags to enable Real-time Audio and MIDI I/O can look like:

```
-+rtmidi=cmidi -M1 -+rtaudio=auhal-odac3 -iadc3
```

# Windows

## Real-time Audio

Windows users can use either the default *PortAudio* Realtime module, or the *winmm* Realtime Audio Module. The winmm module is a native windows module which provides great stability, but latency will usually be too high for realtime interaction. To activate a realtime module, you can use the `-+rtaudio` flag with value of *portaudio* or *winmme*. The default value is *portaudio*, which is activated by default without specifying it.

You also need to specify the sound device you want to use, and specify that you want to generate real-time audio output instead of soundfile to disk output. To do this, you must use the `-odac` or `-o dac` flag, which tells csound to output to the Digital-to-Analog converters instead of a file. By adding a number after the flag (e.g. `-odac2`), you can choose the device number you want. To find out available devices in your system, you can use a large out of range number (e.g. `-odac99`), and csound will report an error, and list available devices.

When choosing the device number under Portaudio, you are also choosing the driver interface, since Portaudio supports WinMME, DirectX and ASIO. If you have an ASIO capable interface or an ASIO driver emulator like ASIO4ALL [<http://www.asio4all.com>], the device will show multiple times, once for each driver interface. ASIO will give you the best latency on your system, so if available it should be your choice for realtime audio output.

Enabling realtime audio input is done using `-iadc`, which makes csound listen to the realtime audio outputs. You can again select the device by its number, and check for available devices using an out of range

number. Note that for input you use 'adc' instead of 'dac'. Make sure you have the appropriate input selected in your soundcard's control panel.

## Real-time MIDI

To enable Real-time MIDI on Windows, you can use the `-M` flag for MIDI input and the `-Q` flag for MIDI output. You might need to specify the device number after the flag (e.g. `-M2`), and again, you can find the available devices by giving an out of range number.

Csound will use PortMidi as the default MIDI module, but there's also a native winmme module, which can be activated with the flag:

```
--rtmidi=winmme
```

A typical set of flags to enable Real-time Audio and MIDI I/O can look like:

```
--rtmidi=winmme -M1 -Q1 --rtaudio=portaudio -odac3 -iadc3
```

## Realtime I/O with JACK Connection Kit

Under a number of systems the JACK connection kit can be used for both audio and MIDI input/output. For more details on this, see

<http://jackaudio.org/faq>

## Using JACK

The simplest way to use the JACK plugin enabling input and output is as follows:

```
--rtaudio=jack -i adc -o dac
```

Additionally, there are some command line options specific to JACK:

### JACK Command-line Flags

`--jack_client=[client_name]`

The client name used by Csound, defaults to 'csound6'. If multiple instances of Csound connect to the JACK server, different client names need to be used to avoid name conflicts.

`--jack_inportname=[input port name prefix]`, `--jack_outportname=[output port name prefix]`

Name prefix of Csound JACK input/output ports; the default is 'input' and 'output'. The actual port name is the channel number appended to the name prefix. Example: with the above default settings, a stereo orchestra will create these ports in full duplex operation:

```
csound6:input1      (record left)
csound6:input2      (record right)
csound6:output1     (playback left)
csound6:output2     (playback right)
```

## Connecting Csound to other JACK clients

By default, connections are made to the first ports on the jack port list (generally these default to system physical ports). The plugin can connect to ports specified as names or numbers.

As names: the port name prefix is used, e.g. "system:playback\_", "system:capture\_", "alsa\_pcm:playback\_" or "alsa\_pcm:capture\_". For example: `-odac:system:playback_` (for output), `-iadc:system:capture_`. The port name prefix excludes channel names.

As numbers: the base port number is given, where the connection is made to this and subsequent ports up to the number of channels. For a base port number  $N$  we will have the connections as  $N+0$ ,  $N+1$ , ...,  $N+nchnls-1$ . For instance `-odac2` with `nchnls=1` will connect outputs to ports 2 and 3. Ports are listed as in other input/output backends.

Note that ports can be prevented from autoconnecting by passing `-odac:null` and `-iadc:null`, for output and input, respectively. Connections can also be made manually using jack connection tools.

## Notes on buffer sizes

Audio data is received from and sent to the JACK server by Csound using a ring buffer that is controlled by the `-b` and `-B` flags. `-B` is the total size of the buffer, while `-b` is the size of a single period. These values are rounded so that the total size is an integer multiple of, and greater than the period size. The difference of the Csound buffer and period size must be greater than or equal to the JACK period size.

If both `-iadc` and `-odac` are used at the same time, the `-b` option should be set to an integer multiple of `ksmps`.

An example of buffer settings for low latency on a fast Linux system:

```
jackd -d alsa -P -r 48000 -p 64 -n 4 -zt &
csound --rtaudio=jack -b 64 -B 256 [...]
```

with real time scheduling (as root):

```
jackd -R -P 90 -d alsa -P -r 48000 -p 64 -n 2 -zt &
csound --sched=80,90,10 -d --rtaudio=jack -b 64 -B 192 [...]
```

To improve performance, use `ksmps` values like 32 and 64.

The sample rate of the orchestra must be the same as that of the JACK server.

Jack can also be used for MIDI IO. For this `--rtmidi=jack` is needed. For input `-M` followed by the requested jack MIDI port name is used to connect directly to an input stream. For output `-Q` followed by the jack port name is used. The options `--jack_midi_inportname=` and `--jack_midi_outportname=` can be used to rename Csound's MIDI IO ports.

## Optimizing Audio I/O Latency

To achieve the lowest latency possible without audio break ups, a combination of variables needs to be tweaked. The final values will be platform and system dependent, and will also depend on the complexity of the audio calculations performed. You need to adjust `ksmps` in the orchestra, as well as the software (`-b`) and hardware buffer (`-B`) sizes.

Usually the simplest solution is the following:

1. Set `ksmps` to a value with a good tradeoff between quality and performance, without adjusting `-B` at all.

2. Set  $-b$  to a negative power of two of this value.

To get the optimal values, start with something you think is going to be too low, ie -1, and then continue "upwards", -2, -4 and so on, until you stop getting x-runs (glitches). The real value of  $-b$  will be the absolute value of  $-b * ksmps$ .

3. Reduce the hardware buffer ( $-B$ ). Bring it down from the default (1024 on Linux, 4096 on Mac OS X, 16384 on Windows), halving it each time, until you start to get x-runs (glitches) again. Then take it back up again until performance is continuous.

This process assumes you have a 16-bit soundcard. If you have a 24-bit soundcard, then  $-B$  should be 3/2, or 3 times  $-b$ , rather than 2 or 4 times. Csound works with 32-bit floats, or 64-bit doubles whereas most soundcards are 16 or 24-bit integer.  $-b$  is the internal buffer, so it's dealing with the 32 or 64-bit side of things, whereas  $-B$  is the hardware buffer, so it's dealing with the 16 or 24-bit side. The csound default for floats is  $-B = 4 * -b$ . This is a sane value for a 16 bit card. You can usually get away with  $-B = 2 * -b$ , but this is the absolute minimum. For example, if you set  $-b1024 -B2048$ , csound will tell you that:

```
audio buffered in 1024 sample-frame blocks
writing 4096-byte blocks to dac
```

4096 bytes is 32768 bits.  $32768/32 = 1024$ , our sample-frame size,  $1024 * 32/16 = 2048$ , our buffer size. Were we to reduce the value of  $-B$ , we would need to reduce the value of  $-b$  by a corresponding amount in order to continue to write 16-bit integers to dac. The minimum size of  $-b$  is  $(-B * bitrate)/32$ . That is to say that the minimum ratio of  $-b$  to  $-B$  should be:

- 16-bit: 1:2
- 24-bit: 2:3
- 32-bit: 1:1

While there is no theoretical maximum ratio, it makes no sense to have a very high ratio here, as the software buffer has to fill the hardware buffer before returning. If the ratio is high, it will take a long time, defeating the purpose of setting a small value for  $-b$ .

The value of  $-b$  is something that will need to be varied depending on the complexity of the instrument you're working with, but because it's intimately related to the value of  $ksmps$ , it's better to synchronise it with  $ksmps$  and go from there. One way to do it is to decide how long the release on your envelopes might need to be at maximum (for desired effect), set the release on all envelopes to maximum, give yourself a generous value for  $-b$ , and then play. If it breaks up, double  $ksmps$ , repeat until smooth, then bring the value of  $-b$  down as far as possible.

The value of  $-B$  is primarily determined by operating system and soundcard. Figure out (using above method) how low you can go, and use that value (or one higher for safety). If you have problems you'll know that it's probably because of an inappropriate value for  $ksmps$ , too low a value for  $-b$ , or denormals (see *denorm*).

---

# Configuring

Once you have either unpacked a binary distribution, or built Csound from sources, you will need to configure Csound so that it will run properly on your system. Installers usually perform these steps for you automatically.

On all platforms, make sure the directory or directories containing Csound's plugin libraries are in an `OPCODE6DIR` or `OPCODE6DIR64` environment variable depending on the precision of the compiled binary. (Note that for `csound5` these environment variables were `OPCODEDIR` and `OPCODEDIR64`.)

The Python opcodes currently require at least Python 2.4, which can be downloaded from [www.python.org](http://www.python.org) [<http://www.python.org>] if it is not already on your system. You can check if it is available by typing 'python' on a command prompt or DOS window.

## Windows

On Windows, make sure the directory or directories (normally the `C:\Program Files\Csound` directory) containing the Csound executables directory are in your `PATH` variable, or else copy all the executable files to your Windows `system32` directory. Depending on your installation method, you might also need to set the `OPCODE6DIR` and `OPCODE6DIR64` environment variables. Assuming that Csound is installed to the default location of `C:\Program Files\Csound` you can use (otherwise set the paths accordingly):

```
set OPCODE6DIR=C:\Program Files\Csound\plugins
set OPCODE6DIR64=C:\Program Files\Csound\plugins64
set PATH=%PATH%;C:\Program Files\Csound\bin
```



### Missing python24.dll or python25.dll

If you get a pop-up about the missing Python library (`python24.dll` or `python25.dll`) and don't need the python opcodes, just delete `C:\Program Files\Csound\plugins\py.dll` and `C:\Program Files\Csound\plugins64\py.dll`, and the pop-up about the missing Python library should be gone.

## Unix and Linux

On Unix and Linux, either install the Csound program in one of the system `bin` directories, typically `/usr/local/bin`, and the Csound and plugin shared libraries in places like `/usr/local/lib/csound/plugins` or `/usr/local/lib/csound/plugins64` and make sure that `OPCODE6DIR` and `OPCODE6DIR64` environment variable are set correctly.

## CsoundAC

CsoundAC requires some additional configuration. On all platforms, CsoundAC requires that you have Python installed on your computer. The directory containing the `_csoundAC` shared library and the `CsoundAC.py` file must be in your `PYTHONPATH` environment variable, so that the Python runtime knows how to load these files.

---

# Syntax of the Orchestra

The Csound orchestra (.orc) or the *<CsInstruments>* section of a csd file, contains:

- A *header section*, which specifies global options for instrument performance
- A list of *User defined opcodes* and *instrument blocks* containing UDO and instrument definitions.

The orchestra header, instrument blocks, and UDOs contain *Orchestra statements*. An *orchestra statement* in Csound has the format:

```
label:  result opcode argument1, argument2, ... ;comments
```

The label is optional and identifies the basic statement that follows as the potential target of a go-to operation (see *Program Flow Control*). A label has no effect on the statement per se.

Depending on their function, some opcodes produce no output, so they have no result value. Others take no arguments and only produce a result.

Every orchestra statement must be on a single line, however long lines can be wrapped to a new line using the '\ ' character. This character indicates that the next line is part of the current one, this way you can split a line for easier reading, like this:

```
a2  oscbnk kcps, 1.0, kfmd1, 0.0, 40, 203, 0.1, 0.2, kamfr, kamfr2, 148, \
    0, 0, 0, 0, 0, 0, -1, \
    kfnum, 3, 4
```

Comments are optional and are for the purpose of letting the user document his orchestra code. Comments begin with a semicolon (;) or // and extend to the end of the line. Comments can optionally be in C-style, spanning multiple lines like this:

```
/* Anything in here -----
   is a comment which can span
   several lines ----- */
```

The remainder (result, opcode, and arguments) form the basic statement. This also is optional, i.e. a line may have only a label or comment or be entirely blank. If present, the basic statement must be complete on one line, and is terminated by a carriage return and line feed.

The opcode determines the operation to be performed; it usually takes some number of input values (or arguments, with a maximum value of about 800); and it usually has a result field variable to which it sends output values at some fixed rate. There are four possible rates:

1. once only, at orchestra setup time (effectively a permanent assignment)
2. once at the beginning of each note (at initialization (init) time: i-rate)
3. once every performance-time control loop (perf-time control rate, or k-rate)
4. once each sound sample of every control loop (perf-time audio rate, or a-rate)

## Orchestra Header Statements

The *Orchestra Header* contains global information that applies to all instruments and defines aspects of Csound output. It is sometimes referred to as *instr 0*, because it behaves as an instrument, but without k- or a-rate processing (i.e. only opcodes and instructions that work at i-rate are allowed).

An *orchestra header statement* operates once only, at orchestra setup time. It is most commonly an assignment of some value to a *global reserved symbol*, e.g. `sr = 20000`. All orchestra header statements belong to a pseudo instrument 0, an *init* pass of which is run prior to all other instruments at score time 0. Any *ordinary statement* can serve as an orchestra header statement, eg. `gifreq = cpspch(8.09)` provided it is an init-time only operation. Statements that are normally placed in an orchestra header are:

- *Odbfs*
- *A4*
- *ctrlinit*
- *ftgen*
- *kr*
- *ksmps*
- *massign*
- *nchnls*
- *pgmassign*
- *pset*
- *seed*
- *sr*
- *strset*

For example, a Csound header may look like:

```
xxsr = 44100
kr = 4410
ksmps = 10
nchnls = 2
odbfs = 1

massign 1, 10
```

## Instrument and Opcode Block Statements

An instrument block is comprised of *ordinary statements* that set values, control the logical flow, or invoke the various signal processing subroutines that lead to audio output. Statements that define an instrument block are:

- *instr*
- *endin*

An instrument block looks like this:

```
instr 1 ;A simple sine wave oscillator
out oscils 10000, 440, 0
out aout
```

*endin*

Statements that define a user defined opcode (UDO) block are

- *opcode*
- *endop*

See the *UDO* section for more information.

## Ordinary Statements

An *ordinary statement* runs at either init time or performance time or both. Operations which produce a result formally run at the rate of that result (that is, at init time for i-rate results; at performance time for k- and a-rate results), with the sole exception of the *init* opcode. Most generators and modifiers, however, produce signals that depend not only on the instantaneous value of their arguments but also on some preserved internal state. These performance-time units therefore have an implicit init-time component to set up that state. The run time of an operation which produces no result is apparent in the opcode.

Arguments are values that are sent to an operation. Most arguments will accept arithmetic expressions composed of constants, variables, reserved symbols, value converters, arithmetic operations, and conditional values.

## Types, Constants and Variables

*Constants* are floating point numbers, such as 1, 3.14159, or -73.45. They are available continuously and do not change in value.

*Variables* are named cells containing numbers. They are available continuously and may be updated at one of the four update rates (setup only, i-rate, k-rate, or a-rate). i- and k-rate variables are scalars (i.e. they take on only one value at any given time) and are primarily used to store and recall controlling data, that is, data that changes at the note rate (for i-rate variables) or at the control rate (for k-rate variables). i- and k-variables are therefore useful for storing note parameter values, pitches, durations, slow-moving frequencies, vibratos, etc. a-rate variables, on the other hand, are arrays or vectors of information. Though renewed on the same perf-time control pass as k-rate variables, these array cells represent a finer resolution of time by dividing the control period into sample periods (see *ksmps*). a-rate variables are used to store and recall data changing at the audio sampling rate (e.g. output signals of oscillators, filters, etc.).

Some types of variables can be thought of as signals. For example a-rate and k-rate variables are signals that have a constant update frequency (see *kr* and *sr*). This abstraction is generally quite useful, but be aware that a-rate signals are actually vectors which are processed at k-rate, i.e. Csound works at k-rate internally but processes *ksmps* number samples for each a-rate variable on every control pass.

There are other types of signals that require rates that don't match *kr* or *sr*. f-rate and w-rate signals are used for spectral processing and their rate is determined by the window size and overlap factor.

A further distinction is that between local and global variables. *local* variables are private to a particular instrument, and cannot be read from or written into by any other instrument. Their values are preserved, and they may carry information from pass to pass (e.g. from initialization time to performance time) within a single instrument. Local variable names begin with the letter *p*, *i*, *k*, or *a*. The same local variable name may appear in two or more different instrument blocks without conflict.

*Global* variables are cells that are accessible by all instruments. The names are either like local names preceded by the letter *g*, or are special reserved symbols. Global variables are used for broadcasting general

values, for communicating between instruments (semaphores), or for sending sound from one instrument to another (e.g. mixing prior to reverberation).

Given these distinctions, there are nine forms of local and global variables:

**Table 3. Types of Variables**

Type	When Renewable	Local	Global
reserved symbols	permanent	--	rsymbol
score pfields	i-time	p number	--
init variables	i-time	i name	gi name
control signals	p-time, k-rate	k name	gk name
audio signals	p-time, k-rate (all audio samples in a k-pass)	a name	ga name
spectral data types	k-rate	w name	--
streaming spectral data types	k-rate	f name	gf name
string variables	i-time and optionally k-rate	S name	gS name
vector variables	k-rate	t name	

Where *rsymbol* is a special reserved symbol (e.g. *sr*, *kr*), *number* is a positive integer referring to a score pfield or sequence number, and *name* is a string of letters, the underscore character, and/or digits with local or global meaning. As might be apparent, score parameters are local i-rate variables whose values are copied from the invoking score statement just prior to the init pass through an instrument, while MIDI controllers are variables which can be updated asynchronously from a MIDI file or MIDI device.

## Variable Initialization

Opcodes that let one initialize variables are:

- *assign*
- *divz*
- *init*
- *tival*

## Predefined Math Constant Macros

Csound defines several important math constants as *Macros*. You can see the full list [here](#).

## Expressions

Expressions may be composed to any depth. Each part of an expression is evaluated at its own proper rate. For instance, if the terms within a sub-expression all change at the control rate or slower, the sub-expression will be evaluated only at the control rate; that result might then be used in an audio-rate evaluation. For example, in

```
k1 + abs(int(p5) + frac(p5) * 100/12 + sqrt(k1))
```

the 100/12 would be evaluated at orch init, the p5 expressions evaluated at note i-time, and the remainder of the expression evaluated every k-period. The whole might occur in a unit generator argument position, or be part of an assignment statement.

## Directories and Files

Many generators and the Csound command itself specify filenames to be read from or written to. These are optionally full pathnames, whose target directory is fully specified. When not a full path, filenames are sought in several directories in order, depending on their type and on the setting of certain environment variables. The latter are optional, but they can serve to partition and organize the directories so that source files can be shared rather than duplicated in several user directories. The environment variables can define directories for soundfiles *SFDIR*, sound samples *SSDIR*, sound analysis *SADIR*, and include files for orchestra and score files *INCDIR*.

In Csound version 5.00 and later, these environment variables can specify multiple directories as a ; separated list. If a file is found in more than one location, the first one has the highest precedence.

The search order is:

1. Soundfiles being written are placed in *SFDIR* (if it exists), else the current directory.
2. Soundfiles for reading are sought in the current directory. If default paths are not disabled, files will next be sought for relative to the CSD/ORC/SCO file. Finally they will be sought in *SSDIR* and then *SFDIR*.
3. Analysis control files for reading are sought in the current directory. If default paths are not disabled, files will next be sought for relative to the CSD/ORC/SCO file. Finally they will be sought in *SADIR*.
4. MIDI files for reading are sought in the current directory. If default paths are not disabled, files will next be sought for relative to the CSD/ORC/SCO file. Finally they will be sought in *MFDIR*, *SSDIR* and *SFDIR*.
5. Files of code to be included in orchestra and score files (with *#include*) are sought first in the current directory, then in the same directory as the orchestra or score file (as appropriate), then finally *INCDIR*.

## Nomenclature

Throughout this document, opcodes are indicated in **boldface** and their argument and result mnemonics, when mentioned in the text, are given in *italics*. Argument names are generally mnemonic (*amp*, *phs*), and the result is usually denoted by the letter *r*. Both are preceded by a type qualifier *i*, *k*, *a*, or *x* (e.g. *kamp*, *iphs*, *ar*). The prefix *i* denotes scalar values valid at note init time; prefixes *k* or *a* denote control (scalar) and audio (vector) values, modified and referenced continuously throughout performance (i.e. at every control period while the instrument is active). Arguments are used at the prefix-listed times; results are created at their listed times, then remain available for use as inputs elsewhere. With few exceptions, argument rates may not exceed the rate of the result. The validity of inputs is defined by the following:

- arguments with prefix *i* must be valid at init time;
- arguments with prefix *k* can be either control or init values (which remain valid);
- arguments with prefix *a* must be vector inputs;
- arguments with prefix *x* may be either vector or scalar (the compiler will distinguish).

All arguments, unless otherwise stated, can be expressions whose results conform to the above. Most opcodes (such as `linen` and `oscil`) can be used in more than one mode, which one being determined by the prefix of the result symbol.

Throughout this manual, the term "opcode" is used to indicate a command that usually produces an a-, k-, or i-rate output, and always forms the basis of a complete Csound orchestra statement. Items such as "+" or "`sin(x)`" or "`( a >= b ? c : d)`" are called "operators."

## Macros

Orchestra macros work like C preprocessor macros, and replace the content of the macro in the orchestra before it is compiled. The opcodes one can use to create, call, or undefine orchestra macros are:

- `#define`
- `$NAME`
- `#ifdef`
- `#ifndef`
- `#end`
- `#else`
- `#include`
- `#undef`

Orchestra macros can also be defined using the command line flag `--omacro:`.

More information and examples on the usage of orchestra macros can be found in the entry for `#define`.

These opcodes refer to orchestra macros; for score macros, refer to *Score Macros*.

## Named Instruments

As a recent addition to the orchestra syntax, instruments can be defined with string names. Such named instruments are callable from the score, and are supported by a number of opcodes.

### Syntax

A named instrument is declared as shown below:

```
instr Name[ , Name2[ , Name3[ , ... ] ]
[ ... ]
endin
```

A single instrument can have any number of names, and any of these names can be used to call the instrument. Additionally, it is possible to use numbers as name, denoting a standard numbered instrument, so the following declaration is also valid:

```
instr 100, Name1, 99, Name2, 1, 2, 3
```

An instrument name may consist of any number of letters, digits, and the underscore (\_) character, however, the first character must not be a digit. Optionally, the instrument name may be prefixed with the '+' character (see below), for example:

```
instr +Reverb
```

For all instrument names, a number is automatically assigned (note: if the message level (-m) is not zero, these numbers are printed to the console during orchestra compilation), following these rules:

- any unused instrument numbers are taken up in ascending order, starting from 1
- the numbers are assigned in the order of instrument name definition, so named instruments that are defined later will always have a higher number (except if the '+' modifier is used)
- if the instrument name was prefixed with '+', the assigned number will be higher than that of any of the (both numbered and named) other instruments without '+'. If there are multiple '+' instruments, the numbering of these will follow the order of definition, according to the above rule.

Using '+' is mainly useful for global output or effect instruments, that must be performed after the other instruments.

An example for instrument numbers:

```
instr 1, 2
endin

instr Instr1
endin

instr +Effect1, Instr2
endin

instr 100, Instr3, +Effect2, Instr4, 5
endin
```

In this example, the instrument numbers are assigned as follows:

```
Instr1: 3
Effect1: 101
Instr2: 4
Instr3: 6
Effect2: 102
Instr4: 7
```

## Using Named Instruments

Named instruments can be called by using the name in double quotes as the instrument number (note: the '+' character should be omitted). Currently (as of Csound 4.22.4), named instruments are supported by:

- 'i' and 'q' score events



### Notes

1. in score files, unmatched quotes, and spaces or other invalid characters in the strings should be avoided, otherwise (at least with current version) unpredictable behavior may

occur (this problem does not exist for -L line events). However, there is checking for undefined instruments, and in such cases, the event is simply ignored with a warning.

2. Stand-alone utilities (score sort and extract) do not support named instruments. It is still possible to sort such scores by using the -t0 option of the main Csound executable)

- real-time line events (-L)
- event, schedkwhen, subinstr, and subinstrinit opcodes
- massign, pgmassign, prealloc, and mute opcodes

Additionally, there is a new opcode (nstrnum) that returns the number of a named instrument:

```
insno nstrnum "name"
```

With the above example, nstrnum "Effect1" would return 101. If an instrument with the specified name does not exist, an init error occurs, and -1 is returned.

## Example

```
; ---- orchestra ----

sr      = 44100
ksmps   = 10
nchnls  = 1

prealloc "SineWave", 20
prealloc "MIDISineWave", 20

massign 1, "MIDISineWave"

gaOutSend      init 0

instr +OutputInstr

out gaOutSend
clear gaOutSend

endin

instr SineWave

a1      oscils p4, p5, 0
vincr gaOutSend, a1

endin

instr MIDISineWave

iamp    veloc
inote   notnum
icps    = cpsoct(inote / 12 + 3)
a1      oscils iamp * 100, icps, 0
vincr gaOutSend, a1

endin

; ---- score ----

i "SineWave" 0 2 12000 440
```

```
i "OutputInstr" 0 3
e
```

## Author

Istvan Varga

2002

## User Defined Opcodes (UDO)

Csound allows the definition of opcodes inside the orchestra header using the opcodes *opcode* and *endop*. The defined opcode may run with a different number of control samples (*ksmps*) using *setksmps*.

To connect inputs and outputs for the UDO, use *xin* and *xout*.

An UDO looks like this:

```
opcode Lowpass, a, akk

    setksmps 1                ; need sr=kr
ain, kal, ka2 xin            ; read input parameters
aout  init 0                 ; initialize output
aout  = ain*kal + aout*ka2    ; simple tone-like filter
xout  aout                   ; write output

endop
```

This UDO called *Lowpass* takes 3 inputs (the first is a-rate, and the next two are k-rate), and delivers 1 a-rate output. Notice the use of *xin* to receive inputs and *xout* to deliver outputs. Also note the use of *setksmps*, which is needed for the filter to work properly.

To use this UDO within an instrument, you would do something like:

```
afiltered Lowpass asource, kvalue1, kvalue2
```

See the entry for *opcode* for detailed information on UDO definition.

You can find many ready made UDO's (or contribute your own) at *Csounds.com* [<http://www.csounds.com/>]'s *User Defined Opcode Database* [<http://www.csounds.com/udo/>].

## K-Rate Vectors

Csound allows the declaration and deployment of one-dimensional vectors or tables. They are local to an instrument, and need to be declared for size (with the *init* opcode. Individual elements are read as part of any expression with square brackets to give an index at k-rate. Individual elements can be assigned, and there are a number of opcodes to query and modify tables.

## Function Syntax in Csound6

Csound 6 has introduced a new alternative syntax for orchestra code. This is initially an experimental feature, which has some limitations, as explained below. It will also allow the introduction of some non-backwards-compatible language features.

## Overview

The main aspect of the new syntax is that some opcodes can be called as functions, and inlined in orchestra code. The general form of this is the expression:

```
var* = op(exprlist*)
```

where \* indicates optional, var is a single variable in one of Csound6 types, and exprlist is a comma separated list of expressions (or a single expression or variable). These expressions can be placed anywhere inside instrument or opcode blocks. I-time operations can also be placed outside instrument blocks. The functional syntax can be intermixed with standard Csound code.

Here are some examples of these expressions:

```
a1 = oscil(p4,p5)
out(vco2(p4*linen(1,0.1,p3,0.1),p5)
outs(oscili(in(),p5), in())
```

## Limitations

The main limitation is that only opcodes with single outputs (or no outputs) are allowed. In addition, opcodes with multiple optional outputs will not be parsed successfully in this form. An alternative to allow for these is to wrap them in user-defined opcodes, or just to intermix standard Csound syntax with this new style.

To resolve opcode ambiguities, we have introduced type annotations, in the form of `op:type(exprlist)`. For instance the code:

```
a1 = oscili(oscili:k(p4,p5), 440)
```

will choose a control-rate opcode to modulate the amplitude of the audio carrier, rather than an audio rate one. There will be cases where the type annotation will be required, when the input arguments cannot be used to determine the correct type of opcode to be applied.

---

# The Standard Numeric Score

The score section contains events that instantiate instruments from the orchestra. There are various score statements that enable complex score building within the csound language.

Currently, the maximum length of the score depends on the platform's architecture; on a 32bit system it is  $2^{31}-1$  control periods; so for example, with  $kr=1500$ , you can run a score for a maximum of about 16.5 days before problems occur due to overflowing signed 32-bit integer variables. On a 64bit machine the same condition would be just about 9 billion years. The input token 'z' is read as a number with the value of approximately 25367 years.

Note also that when using single precision floats (i.e. the 'f' installers rather than the 'd' ones), the accuracy of timing becomes worse after performing for a long time.

## Preprocessing of Standard Scores

A *Score* (a collection of score statements) is divided into time-ordered sections by the *s statement*. Before being read by the orchestra, a score is preprocessed one section at a time. Each section is normally processed by 3 routines: *Carry*, *Tempo*, and *Sort*.

### Carry

Within a group of consecutive *i statements* whose p1 whole numbers correspond, any pfield left empty will take its value from the same pfield of the preceding statement. An empty pfield can be denoted by a single point (.) delimited by spaces. No point is required after the last nonempty pfield. The output of Carry preprocessing will show the carried values explicitly. The Carry Feature is not affected by intervening comments or blank lines; it is turned off only by a non- *i statement* or by an *i statement* with unlike p1 whole number.

Three additional features are available for p2 alone: +, ^+x, and ^-x. The symbol + in p2 will be given the value of p2 + p3 from the preceding *i statement*. This enables note action times to be automatically determined from the sum of preceding durations. The + symbol can itself be carried. It is legal only in p2. E.g.: the statements

```
i1 0 .5 100
i . +
i
```

will result in

```
i1 0 .5 100
i1 .5 .5 100
i1 1 .5 100
```

The symbols ^+x and ^-x determine the current p2 by adding or subtracting, respectively, the value of x from the preceding p2. These may be used in p2 only and are *not* carried like the + symbol. Note also that there should be no spaces following the ^, the +, or the - parts of these symbols -- the number must come directly after as in ^+2.3. If the example above had been

```
i1 0 .5 100
i . ^+1
i . ^+1
```

the result would instead be

```
i1 0      .5      100
i1 1      .5      100
i1 2      .5      100
```

The Carry feature should be used liberally. Its use, especially in large scores, can greatly reduce input typing and will simplify later changes.

There can sometimes be circumstances where you do not want "missing" pfields after the last one entered to be implicitly carried. An example would be an instrument that is designed to take a variable number of pfields. Beginning with Csound 5.08, you can prevent the implicit carrying of pfields at the end of an *i* statement by using the symbol ! (called the "no-carry symbol"). The ! must appear at the end of an *i* statement and it cannot be used in p1, p2, or p3, since these pfields are required. Here is an example:

```
i1 0      .5      100
i . +
i . . . !
i
```

This score would be interpreted as

```
i1 0      .5      100
i1 .5     .5      100
i1 1      .5
i1 1.5    .5      ; no p4
                    ; only p1 to p3 are carried here
```

An alternative to using ! is to switch automatic carrying off apart from p1, p2 and p3. This can be done with the score opcode statement "C 0", and can be restored with "C 1".

## Tempo

This operation time warps a score section according to the information in a *t statement*. The tempo operation converts p2 (and, for *i statements*, p3) from original beats into real seconds, since those are the units required by the orchestra. After time warping, score files will be seen to have orchestra-readable format demonstrated by the following:

```
i p1 p2beats p2seconds p3beats p3seconds p4 p5 ....
```

## Sort

This routine sorts all action-time statements into chronological order by p2 value. It also sorts coincident events into precedence order. Whenever an *f statement* and an *i statement* have the same p2 value, the *f statement* will precede. Whenever two or more *i statements* have the same p2 value, they will be sorted into ascending p1 value order. If they also have the same p1 value, they will be sorted into ascending p3 value order. Score sorting is done section by section (see *s statement*). Automatic sorting implies that score statements may appear in any order within a section.



### Note

The operations Carry, Tempo and Sort are combined in a 3-phase single pass over a score file, to produce a new file in orchestra-readable format ( see the Tempo example). Processing can

be invoked either explicitly by the *Scsort* command, or implicitly by *Csound* which processes the score before calling the orchestra. Source-format files and orchestra-readable files are both in ASCII character form, and may be either perused or further modified by standard text editors. User-written routines can be used to modify score files before or after the above processes, provided the final orchestra-readable statement format is not violated. Sections of different formats can be sequentially batched; and sections of like format can be merged for automatic sorting.

## Score Statements

The statements used in scores are:

- *a* - Advance score time by a specified amount
- *b* - Resets the clock
- *C* - Toggles carry facility
- *d* - Deletes an infinite instrument
- *e* - Marks the end of the last section of the score
- *f* - Causes a *GEN subroutine* to place values in a stored function table
- *i* - Makes an instrument active at a specific time and for a certain duration
- *m* - Sets a named mark in the score
- *n* - Repeats a section
- *q* - Used to quiet an instrument
- *r* - Starts a repeated section
- *s* - Marks the end of a section
- *t* - Sets the tempo
- *v* - Provides for locally variable time warping of score events
- *x* - Skip the rest of the current section
- *y* - Set seed for random numbers, either from *p1* or, if omitted, the clock
- *{* - Begins a non-sectional, nestable loop.
- *}* - Ends a non-sectional, nestable loop.

Comments are denoted by semicolon (;), double slash (//) or the character *c* and last until a newline. Also C-style comments */\* ... \*/* are allowed.

## Next-P and Previous-P Symbols

At the close of any of the operations *Carry*, *Tempo*, and *Sort*, three additional score features are interpreted during file writeout: *next-p*, *previous-p*, and *ramping*.

*i statement* pfields containing the symbols *np<sub>x</sub>* or *pp<sub>x</sub>* (where *x* is some integer) will be replaced by the appropriate pfield value found on the next *i statement* (or previous *i statement*) that has the same p1. For example, the symbol *np7* will be replaced by the value found in p7 of the next note that is to be played by this instrument. *np* and *pp* symbols are recursive and can reference other *np* and *pp* symbols which can reference others, etc. References must eventually terminate in a real number or a *ramp symbol*. Closed loop references should be avoided. *np* and *pp* symbols are illegal in p1, p2 and p3 (although they may reference these). *np* and *pp* symbols may be Carried. *np* and *pp* references cannot cross a Section boundary. Any forward or backward reference to a non-existent note-statement will be given the value zero.

E.g.: the statements

```
i1  0  1  10  np4  pp5
i1  1  1  20
i1  1  1  30
```

will result in

```
i1  0  1  10  20  0
i1  1  1  20  30  20
i1  2  1  30  0  30
```

*np* and *pp* symbols can provide an instrument with contextual knowledge of the score, enabling it to glissando or crescendo, for instance, toward the pitch or dynamic of some future event (which may or may not be immediately adjacent). Note that while the *Carry* feature will propagate *np* and *pp* through unsorted statements, the operation that interprets these symbols is acting on a fully sorted version of the score. The tempo operation is applied after the *pp* and/or *np* processing.

## Ramping

*i statement* pfields containing the symbol *<* will be replaced by values derived from linear interpolation of a time-based ramp. Ramps are anchored at each end by the first real number found in the same pfield of a preceding and following note played by the same instrument. E.g.: the statements

```
i1  0  1  100
i1  1  1  <
i1  2  1  <
i1  3  1  400
i1  4  1  <
i1  5  1  0
```

will result in

```
i1  0  1  100
i1  1  1  200
i1  2  1  300
i1  3  1  400
i1  4  1  200
i1  5  1  0
```

Ramps cannot cross a Section boundary. Ramps cannot be anchored by an *np* or *pp* symbol (although they may be referenced by these). Ramp symbols are illegal in p1, p2 and p3. Ramp symbols may be Carried. Note, however, that while the *Carry* feature will propagate ramp symbols through unsorted statements, the operation that interprets these symbols is acting on a time-warped and fully sorted version of the score. In fact, time-based linear interpolation is based on warped score-time, so that a ramp which spans a group of accelerating notes will remain linear with respect to strict chronological time.

Starting with Csound version 3.52, using the symbols ( or ) will result in an exponential interpolation ramp, similar to *expon*. Using the symbol ~ (a tilde) will result in uniform, random distribution between the first and last values of the ramp. Use of these functions must follow the same rules as the linear ramp function.

## Score Macros

### Description

Macros are textual replacements which are made in the score as it is being presented to the system. The macro system in Csound is a very simple one, and uses the characters # and \$ to define and call macros. This can allow for simpler score writing, and provide an elementary alternative to full score generation systems. The score macro system is similar to, but independent of, the macro system in the orchestra language.

*#define* NAME -- defines a simple macro. The name of the macro must begin with a letter and can consist of any combination of letters and numbers. Case is significant. This form is limiting, in that the variable names are fixed. More flexibility can be obtained by using a macro with arguments, described below.

*#define* NAME(a' b' c') -- defines a macro with arguments. This can be used in more complex situations. The name of the macro must begin with a letter and can consist of any combination of letters and numbers. Within the replacement text, the arguments can be substituted by the form: \$A. In fact, the implementation defines the arguments as simple macros. There may be up to 5 arguments, and the names may be any choice of letters. Remember that case is significant in macro names.

*\$NAME.* -- calls a defined macro. To use a macro, the name is used following a \$ character. The name is terminated by the first character which is neither a letter nor a number. If it is necessary for the name not to terminate with a space, a period, which will be ignored, can be used to terminate the name. The string, *\$NAME.*, is replaced by the replacement text from the definition. The replacement text can also include macro calls.

*#undef* NAME -- undefines a macro name. If a macro is no longer required, it can be undefined with *#undef* NAME.

### Syntax

```
#define NAME # replacement text #  
  
#define NAME(a' b' c') # replacement text #  
  
$NAME.  
  
#undef NAME
```

### Initialization

*# replacement text #* -- The replacement text is any character string (not containing a #) and can extend over multiple lines. The replacement text is enclosed within the # characters, which ensure that additional characters are not inadvertently captured.

### Performance

Some care is needed with textual replacement macros, as they can sometimes do strange things. They take no notice of any meaning, so spaces are significant. This is why, unlike the C programming language, the definition has the replacement text surrounded by # characters. Used carefully, this simple macro system is a powerful concept, but it can be abused.

**Another Use For Macros.** When writing a complex score it is sometimes all too easy to forget to what the various instrument numbers refer. One can use macros to give names to the numbers. For example

```
#define Flute #i1#
#define Whoop #i2#

$Flute. 0 10 4000 440
$Whoop. 5 1
```

## Examples

### Example 1. Simple Macro

A note-event has a set of p-fields which are repeated:

```
#define ARGS # 1.01 2.33 138#
i1 0 1 8.00 1000 $ARGS
i1 0 1 8.01 1500 $ARGS
i1 0 1 8.02 1200 $ARGS
i1 0 1 8.03 1000 $ARGS
```

This will get expanded before sorting into:

```
i1 0 1 8.00 1000 1.01 2.33 138
i1 0 1 8.01 1500 1.01 2.33 138
i1 0 1 8.02 1200 1.01 2.33 138
i1 0 1 8.03 1000 1.01 2.33 138
```

This can save typing, and it makes revisions easier. If there were two sets of p-fields one could have a second macro (there is no real limit on the number of macros one can define).

```
#define ARGS1 # 1.01 2.33 138#
#define ARGS2 # 1.41 10.33 1.00#
i1 0 1 8.00 1000 $ARGS1
i1 0 1 8.01 1500 $ARGS2
i1 0 1 8.02 1200 $ARGS1
i1 0 1 8.03 1000 $ARGS2
```

### Example 2. Macros with arguments

```
#define ARG(A) # 2.345 1.03 $A 234.9#
i1 0 1 8.00 1000 $ARG(2.0)
i1 + 1 8.01 1200 $ARG(3.0)
```

which expands to

```
i1 0 1 8.00 1000 2.345 1.03 2.0 234.9
i1 + 1 8.01 1200 2.345 1.03 3.0 234.9
```

## Credits

Author: John ffitch

University of Bath/Codemist Ltd.

Bath, UK

April, 1998 (New in Csound version 3.48)

# Multiple File Score

## Description

Using the score in more than one file.

## Syntax

```
#include "filename"
```

## Performance

It is sometimes convenient to have the score in more than one file. This use is supported by the *#include* facility which is part of the macro system. A line containing the text

```
#include "filename"
```

where the character " can be replaced by any suitable character. For most uses the double quote symbol will probably be the most convenient. The file name can include a full path.

This takes input from the named file until it ends, when input reverts to the previous input. There is currently a limit of 20 on the depth of included files and macros.

A suggested use of *#include* would be to define a set of macros which are part of the composer's style. It could also be used to provide repeated sections.

```
S
#include :section1:
;; Repeat that
S
#include :section1:
```

Alternative methods of doing repeats, use the *r statement*, *m statement*, and *n statement*.

## Credits

Author: John ffitch

University of Bath/Codemist Ltd.

Bath, UK

April, 1998 (New in Csound version 3.48)

Thanks to Luis Jure for pointing out the incorrect syntax in multiple file include statement.

## Evaluation of Expressions

In earlier versions of Csound the numbers presented in a score were used as given. There are occasions when some simple evaluation would be easier. This need is increased when there are macros. To assist in

this area the syntax of arithmetic expressions within square brackets [ ] has been introduced. Expressions built from the operations +, -, \*, /, % ("modulo"), and ^ ("power of") are allowed, together with grouping with (). Unary minus and plus are also supported. The expressions can include numbers, and naturally macros whose values are numeric or arithmetic strings. All calculations are made in floating point numbers. The usual precedence rules are followed when evaluating: expressions within parantheses () are evaluated first and ^ is evaluated before \*, /, and % which are evaluated before + and -.

In addition to arithmetic operations, the following bitwise logical operators are also available: & (AND), | (OR), and # (XOR, exclusive-OR). These operators round their operands to the nearest (long) integer before evaluating. The logical operators have the same precedence as the \*, /, and % arithmetic operators.

Finally, the tilde symbol ~ can be used in an expression wherever a number is permissible to use. Each ~ will evaluate to a random value between zero (0) and one (1).

## Example

```
r3 CNT
i1 0 [0.3*$CNT.]
i1 + [($CNT./3)+0.2]
e
```

As the three copies of the section have the macro \$CNT. with the different values of 1, 2 and 3, this expands to

```
s
i1 0 0.3
i1 0.3 0.533333
s
i1 0 0.6
i1 0.6 0.866667
s
i1 0 0.9
i1 0.9 1.2
e
```

This is an extreme form, but the evaluation system can be used to ensure that repeated sections are subtly different.

Here are some simple examples of each operator:

```
i1 0 1 [ 110 + 220 ] ; evaluates to 330
i1 + . [ 330 - 55 ] ; 275
i1 + . [ 44 * 10 ] ; 440
i1 + . [ 1100 / 2 ] ; 550
i1 + . [ 5 ^ 4 ] ; 625
i1 + . [ 5660 % 1000 ] ; 660
i1 + . [ 110 & 220 ] ; 76
i1 + . [ 110 | 220 ] ; 254
i1 + . [ 110 # 220 ] ; 178
i1 + . [~] ; random between 0-1
i1 + . [~ * 4 + 1] ; random between 1-5
i1 + . [~ * 95 + 5] ; random between 5-100

i1 + . [ 8 / 2 * 3 ] ; 12
i1 + . [ 4 + 3 - 2 + 1 ] ; 6
i1 + . [ 4 + 3 * 2 + 1 ] ; 11
i1 + . [(4 + 3)*(2 + 1)] ; 21
```

```
i1 + . [ 2 * 2 & 3 ] ; 4
i1 + . [ 3 & 2 * 2 ] ; 0
i1 + . [ 4 | 3 * 3 ] ; 13
```

## The @ operator

New in Csound version 3.56 are `@x` (next power-of-two greater than or equal to  $x$ ) and `@@x` (next power-of-two-plus-one greater than or equal to  $x$ ).

```
[ @ 11 ] will evaluate to 16
[ @@ 11 ] to 17
```

## Credits

Author: John ffitch

University of Bath/Codemist Ltd.

Bath, UK

April, 1998 (New in Csound version 3.48)

## Strings in p-fields

You can pass a string as a p-field instead of a number, like this:

```
i 1 0 10 "A4"
```

The string can be received by the instrument and further processed using the *string opcodes*.



### Note

Currently only one p-field can contain a string (i.e. no more than one string per line is allowed). You can overcome this using *strset* and *strget*.

---

# Front Ends

Front ends are programs that provide some form of user interface for Csound. Within these programs, Csound is used to generate sound, and familiarity with Csound code is required in order to use them. Front ends typically add helpful features, such as syntax coloring, graphic widgets, or tools for algorithmic score generation, that are not part of Csound itself. Most of these programs were created by a single person, so some of them are not being maintained. Below is a list (certainly not complete, and perhaps not up to date) of front ends available for Csound.

Most often, you'll want to download and install Csound itself before downloading and installing a front end. Some front ends require particular versions of Csound, so if you plan to use a front end, it's recommended that you verify its compatibility before installing Csound.

## CsoundQt

CsoundQt is a versatile, cross-platform GUI (graphical user interface) which is bundled with the standard Csound distribution. Created and maintained by Andres Cabrera, QuteCsound provides a multi-tabbed editor, graphic widgets for real-time sound control, and an opcode help system that links to this manual. At this writing (2013) CsoundQt is in active development, so the version installed in your system when you install Csound may not be the most current. The most recent version can be found at <http://qute-csound.sourceforge.net/>.

## Blue

A cross-platform composition-oriented front end written by Steven Yi in Java. The user interface provides a timeline structured somewhat like a digital multitrack, but differs in that timelines can be embedded within timelines (polyObjects). This allows for a compositional organization in time that many users will find intuitive, informative, and flexible. Each instrument and score section in a blue project has its own editing window, which makes organizing large projects easier. Blue can be downloaded at Blue Home Page [<http://csounds.com/stevenyi/blue/>].

## Cabbage

Cabbage is a Csound frontend that provides users with the means to develop audio plugins and standalone software across the three major operating systems. While Cabbage makes use of underlying plugin technologies such as Steinberg's VST SDK, ASIO, etc, Csound is used to process all incoming and outgoing audio. Cabbage also provides a growing collection of GUI widgets ranging from simple sliders to automatable XY-pads. All GUI widgets in a Cabbage plugin can be controlled via host automation in a plugin host, thereby providing a quick and effective means of automating Csound instrument parameters in both commercial and non-commercial DAWs. Cabbage can be downloaded at Cabbage Home Page [<https://github.com/cabbageaudio/cabbage/releases>].

## WinXound

WinXound is a free and open-source Front-End GUI Editor with syntax highlighting for CSound 6, CSoundAV, CSoundAC, with Python and Lua support, developed by Stefano Bonetti. It runs on Microsoft Windows, Apple Mac OsX and Linux. You can get it at the WinXound Front Page [<http://winxound.codeplex.com/>].

## Winsound

Winsound was formerly part of the main Csound tree. It is now available only as source code. Winsound is a cross-platform FLTK port of Barry Vercoe's original front-end for csound. Some partially sighted or unsighted users report success using Winsound with text-to-speech software.

## CsoundAC

### Python Scripting

You can use CsoundAC as a Python extension module. You can do this in a standard Python interpreter, such as Python command line or the Idle Python GUI.

To use CsoundAC in a standard Python interpreter, import CsoundAC.

```
import CsoundAC
```

The CsoundAC module automatically creates an instance of CppSound named `csound`, which provides an object-oriented interface to the Csound API. In a standard Python interpreter, you can load a Csound `.csd` file and perform it like this:

```
C:\Documents and Settings\mkg>python
Python 2.3.3 (#51, Dec 18 2003, 20:22:39) [MSC v.1200 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import CsoundAC
>>> csound.load("c:/projects/csound5/examples/trapped.csd")
1
>>> csound.exportForPerformance()
1
>>> csound.perform()
BEGAN CppSound::perform(5, 988ee0)...
BEGAN CppSound::compile(5, 988ee0)...
Using default language
0dBFS level = 32767.0
Csound version 5.00 beta (float samples) Jun  7 2004
libsndfile-1.0.10pre6
orchname: temp.orc
scorename: temp.sco
orch compiler:
398 lines read
instr 1
instr 2
instr 3
instr 4
instr 5
instr 6
instr 7
instr 8
instr 9
instr 10
instr 11
instr 12
instr 13
instr 98
instr 99
sorting score ...
... done
Csound version 5.00 beta (float samples) Jun  6 2004
displays suppressed
0dBFS level = 32767.0
```

```

orch now loaded
audio buffered in 16384 sample-frame blocks
SFDIR undefined. using current directory
writing 131072-byte blks of shorts to test.wav
WAV
SECTION 1:
ENDED CppSound::compile.
ftable 1:
ftable 2:
ftable 3:
ftable 4:
ftable 5:
ftable 6:
ftable 7:
ftable 8:
ftable 9:
ftable 10:
ftable 11:
ftable 12:
ftable 13:
ftable 14:
ftable 15:
ftable 16:
ftable 17:
ftable 18:
ftable 19:
ftable 20:
ftable 21:
ftable 22:
new alloc for instr 1:
B 0.000 .. 1.000 T 1.000 TT 1.000 M: 32.7 0.0
new alloc for instr 1:
B 1.000 .. 3.600 T 3.600 TT 3.600 M: 207.6 0.1
...

B 93.940 .. 94.418 T 98.799 TT281.799 M: 477.6 85.0
B 94.418 ..100.000 T107.172 TT290.172 M: 118.9 11.5
end of section 4 sect peak amps: 25950.8 26877.4
inactive allocs returned to freespace
end of score. overall amps: 32204.8 31469.6
overall samples out of range: 0 0
0 errors in performance
782 131072-byte soundblks of shorts written to test.wav WAV
Elapsed time = 13.469000 seconds.
ENDED CppSound::perform.
1
>>>

```

The `koch.py` script shows how to use Python to do algorithmic composition for Csound. You can use Python triple-quoted string literals to hold your Csound files right in your script, and assign them to Csound:

```

csound.setOrchestra('''sr = 44100
kr = 441
ksmps = 100
nchnls = 2
0dbfs = .1
instr 1,2,3,4,5 ; FluidSynth General MID
I; INITIALIZATION
; Channel, bank, and program determine the preset, that is, the actual sound.
ichannel = p1
iprogram = p6
ikey = p4
ivelocity = p5 + 12

```

```
ijunk6    =    p6
ijunk7    =    p7
; AUDIO
istatus   =    144;
print    iprogram, istatus, ichannel, ikey, ivelocityleft, aright
fluid    "c:/projects/csound5/samples/VintageDreamsWaves-v2.sf2", \\
iprogram, istatus, ichannel, ikey, ivelocity, 1
outs     aleft, arightendin''')
csound.setCommand("csound --opcode-lib=c:/projects/csound5/fluid.dll \\
-RWdfo ./koch.wav ./temp.orc ./temp.sco")
csound.exportForPerformance()
csound.perform()
```

## CsoundVST

CsoundVST is a multi-function front end for Csound, based on the Csound API. CsoundVST runs as a stand-alone graphical user interface to Csound, and it also runs as a VST instrument or effect plugin in VST hosts such as Cubase with the same user interface. CsoundVST is part of the main csound source tree, but is not included in standard distributions, due to licensing limitations of Steinberg's VST SDK.

## Standalone

To run CsoundVST as a stand-alone front end to Csound, execute CsoundVST. When the program has loaded, you will see a graphical user interface with a row of buttons along the top. Click on the *Open...* button to load a `.csd` file. You can also click on the *Open...* button and load a `.orc` file, then click on the *Import...* button to add a `.sco` file. You can edit the Csound command, the orchestra file, or the score file in the respective tabs of the user interface. When all is satisfactory, click on the *Perform* button to run Csound. You can stop a performance at any time by clicking on the *Stop* button.

## VST Plugin

The following instructions are for Cubase 4.0. You would follow roughly similar procedures in other hosts.

Use the *Devices* menu, *Plug-In Information* dialog, *VST Plug-Ins* tab, *VST 2.x Plug-in Paths* dialog, *Add* button to add your `csound/bin` directory to Cubase's plugin path. You can have multiple directories separated by semicolons. Then select the CsoundVST path and click on the *Set as Shared Folder* button.

Quit Cubase, and start it again.

Use the *File* menu, *New Project* dialog to create a new song.

Use the *Project* menu, *Add Track* submenu, to add a new MIDI track.

Use the pencil tool to draw a *Part* on the track a few measures long. Write some music in the *Part* using the *Event* editor or the *Score* editor.

Use the *Devices* menu (or the F11 key) to open the *VST Instruments* dialog.

Click on one of the *No VST Instrument* labels, and select *CsoundVST* from the list that pops up.

Click on the *e* (for edit) button to open the *CsoundVST* dialog.

On the Settings page, check the *Instrument* box in the VST Plugin group, and the *Classic* box in the *Csound performance mode* group. Then click on the *Apply* button.

Click on the *Open* button to bring up the file selector dialog. Navigate to a directory containing a Csound `csd` file suitable for MIDI performance, such as `csound/examples/CsoundVST.csd`. Click on the OK button to load the file. You can also open and import a suitable `.orc` and `.sco` file as described above.

In any event, the command line in the *Classic Csound command line* text box must specify `--rtmidi=null -M0`, and should read something like this:

```
csound -f -h --rtmidi=null -M0 -d -n -m7 --midi-key-oct=4 --midi-velocity=5 temp.orc temp.sco
```

Click on the *VST Instruments* dialog's on/off button to turn it on. This should compile the Csound orchestra.

In the *Cubase Track Inspector*, click on the *out: Not Assigned* label and select *CsoundVST* from the list that pops up.

On the ruler at the top of the *Arrangement* window, select the loop end point and drag it to the end of your part, then click on the loop button to enable looping.

Click on the *play* button on the *Transport* bar. You should hear your music played by CsoundVST.

Try assigning your track to different channels; a different Csound instrument will perform each channel.

When you save your song, your Csound orchestra will be saved as part of the song and re-loaded when you re-load the song.

You can click on the *Orchestra* tab and edit your Csound instruments while CsoundVST is playing. To hear your changes, just click on the CsoundVST *Perform* button to recompile the orchestra.

You can assign up to 16 channels to a single CsoundVST plugin.

---

# Building Csound

Csound has become a complex project and can involve many dependencies. Unless you are a Csound developer or need to develop Csound plugins, you should try to use one of the precompiled distributions from <http://www.sourceforge.net/projects/csound>.

Detailed and up to date information about building Csound from source can be found in the BUILD.md [<https://github.com/csound/csound/blob/develop/BUILD.md>] file in the Csound6 sources.

---

# Csound Links

Csound's "home page" can be found at <http://csound.github.io>.

Another Csound page, maintained by Richard Boulanger, can be found at <http://csounds.com>.

The Csound source code is maintained by John ffitch and others at <https://github.com/csound>. The most recent versions and precompiled packages for most platforms also can be downloaded here [[http://sourceforge.net/project/showfiles.php?group\\_id=81968](http://sourceforge.net/project/showfiles.php?group_id=81968)].

A Csound mailing list exists to discuss Csound. It is run by John ffitch and Victor Lazzarini of Maynooth University, Ireland. To have your name put on the mailing list send a message to: [listserv@heanet.ie](mailto:listserv@heanet.ie) [<mailto:listserv@listserv.heanet.ie>] with body "subscribe csound". You can also subscribe to the digest (1 message per day) by sending an email to: [listserv@listserv.heanet.ie](mailto:listserv@listserv.heanet.ie) [<mailto:listserv@listserv.heanet.ie>] with body "subscribe csound set digest". Posts sent to [csound@listserv.heanet.ie](mailto:csound@listserv.heanet.ie) [<mailto:csound@listserv.heanet.ie>] go to all subscribed members of the list.

Similarly, the `csound-devel` mailing list exists to discuss Csound development. For more information on this list, go to <http://listserv.heanet.ie> [<http://listserv.heanet.ie/>] and follow the link to `csound-dev`. Posts sent to [csound-dev@listserv.heanet.ie](mailto:csound-dev@listserv.heanet.ie) [<mailto:csound-dev@listserv.heanet.ie>] go to all subscribed members of the list.

Suspected bugs in the code may be entered using the bug tracking system at the github [<https://github.com/csound/csound/issues>].

---

# Part II. Opcodes Overview

---

---

# Table of Contents

Signal Generators .....	82
Additive Synthesis/Resynthesis .....	82
Basic Oscillators .....	82
Dynamic Spectrum Oscillators .....	82
FM Synthesis .....	83
Granular Synthesis .....	83
Hyper Vectorial Synthesis .....	84
Linear and Exponential Generators .....	84
Envelope Generators .....	85
Models and Emulations .....	85
Phasors .....	86
Random (Noise) Generators .....	87
Sample Playback .....	88
Soundfonts .....	88
Scanned Synthesis .....	90
Table Access .....	91
Wave Terrain Synthesis .....	92
Waveguide Physical Modeling .....	92
Signal Input and Output .....	93
File Input and Output .....	93
Signal Input .....	93
Signal Output .....	93
Software Bus .....	94
Printing and Display .....	94
Sound File Queries .....	94
Signal Modifiers .....	96
Amplitude Modifiers and Dynamic processing .....	96
Convolution and Morphing .....	96
Delay .....	96
Panning and Spatialization .....	97
Reverberation .....	98
Sample Level Operators .....	99
Signal Limiters .....	99
Special Effects .....	99
Standard Filters .....	100
Specialized Filters .....	102
Waveguides .....	102
Waveshaping and Phase Distortion .....	102
Instrument Control .....	104
Clock Control .....	104
Conditional Values .....	104
Duration Control Statements .....	104
FLTK Widgets and GUI controllers .....	104
FLTK Containers .....	107
FLTK Valuators .....	107
Other FLTK Widgets .....	108
Modifying FLTK Widget Appearance .....	108
General FLTK Widget-related Opcodes .....	109
Instrument Invocation .....	109
Program Flow Control .....	110
Real-time Performance Control .....	111

Initialization and Reinitialization .....	111
Sensing and Control .....	112
Stacks .....	113
Sub-instrument Control .....	113
Time Reading .....	113
Function Table Control .....	115
Table Queries .....	115
Read/Write Operations .....	115
Table Reading with Dynamic Selection .....	116
Mathematical Operations .....	117
Amplitude Converters .....	117
Arithmetic and Logic Operations .....	117
Comparators and Accumulators .....	117
Mathematical Functions .....	118
Opcode Equivalents of Functions .....	118
Random Functions .....	118
Trigonometric Functions .....	119
Linear Algebra Opcodes .....	120
Array Opcodes .....	128
Pitch Converters .....	135
Functions .....	135
Tuning Opcodes .....	135
Real-time MIDI Support .....	136
Virtual MIDI Keyboard .....	137
MIDI input .....	140
MIDI Message Output .....	140
Generic Input and Output .....	141
Converters .....	141
Event Extenders .....	141
Note-on/Note-off Output .....	141
MIDI/Score Interoperability opcodes .....	142
System Realtime Messages .....	143
Slider Banks .....	143
Spectral Processing .....	144
Short-time Fourier Transform (STFT) Resynthesis .....	144
Linear Predictive Coding (LPC) Resynthesis .....	144
Non-standard Spectral Processing .....	145
Tools for Real-time Spectral Processing (pvs opcodes) .....	145
ATS Spectral Processing .....	146
Loris Opcodes .....	147
Array-based spectral opcodes .....	150
Strings .....	152
String Manipulation Opcodes .....	153
String Conversion Opcodes .....	154
Vectorial Opcodes .....	155
Tables of vectors operators .....	155
Operations Between a Vectorial and a Scalar Signal .....	155
Operations Between two Vectorial Signals .....	156
Vectorial Envelope Generators .....	156
Limiting and wrapping of vectorial control signals .....	157
Vectorial Control-rate Delay Paths .....	157
Vectorial Random Signal Generators .....	157
Zak Patch System .....	158
Plugin Hosting .....	159

DSSI and LADSPA for Csound .....	159
VST for Csound .....	159
OSC and Network .....	161
Ableton Link Opcodes .....	161
OSC .....	161
Network .....	162
Remote Opcodes .....	162
Mixer Opcodes .....	163
Signal Flow Graph Opcodes .....	164
Jacko Opcodes .....	167
Lua Opcodes .....	170
Python Opcodes .....	175
Introduction .....	175
Orchestra Syntax .....	175
Image processing opcodes .....	177
STK Opcodes .....	178
Miscellaneous opcodes .....	180

---

# Signal Generators

## Additive Synthesis/Resynthesis

The opcodes for additive synthesis and resynthesis are:

- *adsyn*
- *adsynt*
- *adsynt2*
- *hsboscil*

See the section *Spectral processing* for more information and further additive/resynthesis opcodes.

## Basic Oscillators

The basic oscillator opcodes are: (note that opcodes that end with 'i' implement linear interpolation and those that end with '3' implement cubic interpolation)

- Oscillator Banks: *oscbnk*
- Simple table oscillators: *oscil*, *oscil3* and *oscili*.
- Simple, fast sine oscillator: *oscils*
- Precision oscillators: *poscil* and *poscil3*.
- More flexible oscillators: *oscilikt*, *osciliktp*, *oscilikts* and *osciln* (also called *oscilx*).

Oscillators can also be constructed from generic table read opcodes. See the *Table Read/Write operations* section.

## LFOs

- *lfo*
- *vibr*
- *vibrato*

See the section *Table access* for other table reading opcodes that can be used as oscillators. Also see the section *Dynamic spectrum Oscillators*.

## Dynamic Spectrum Oscillators

The opcodes that generate dynamic spectra are:

- Harmonic spectra: *buzz* and *gbuzz*
- Impulse generator: *mpulse*

- Band limited oscillators (analog modelled): *vco* and *vco2*

The following opcodes can be used to generate band-limited waveforms for use with *vco2* and other oscillators:

- *vco2init*
- *vco2ft*
- *vco2ift*

## FM Synthesis

The FM synthesis opcodes are:

- *foscil*
- *foscili*
- *crossfm*, *crossfmi*, *crosspm*, *crosspmi*, *crossfmpm*, and *crossfmpmi*.

## FM instrument models

- *fmb3*
- *fmbell*
- *fmmetal*
- *fmpercfl*
- *fmrhode*
- *fmvoice*
- *fmwurlie*

## Granular Synthesis

The granular synthesis opcodes are:

- *diskgrain*
- *fof*
- *fof2*
- *fog*
- *grain*
- *grain2*
- *grain3*
- *granule*

- *partikkel*
- *partikkelsync*
- *sndwarp*
- *sndwarpst*
- *syncgrain*
- *syncloop*
- *vosim*

## Hyper Vectorial Synthesis

- *vphaseseg*
- *hvs1*
- *hvs2*
- *hvs3*

## Linear and Exponential Generators

The opcodes that generate linear or exponential curves or segments are:

- *expon*
- *expcurve*
- *expseg*
- *expsega*
- *expsegr*
- *gainslider*
- *jspline*
- *line*
- *linseg*
- *linsegr*
- *logcurve*
- *loopseg*
- *loopsegp*
- *lpshold*
- *lpsholdp*

- *rspline*
- *scale*
- *transeg*
- *bpf*
- *linlin*
- *xyscale*

## Envelope Generators

The following envelope generators are available:

- *adsr*
- *madsr*
- *mxadsr*
- *xadsr*
- *linen*
- *linenr*
- *envlpx*
- *envlpxr*
- *lineto*
- *tilineto*

Consult the *Linear and exponential generators* section for additional methods to create envelopes.

## Models and Emulations

The following opcodes model or emulate the sounds of other instruments (some based on the STK toolkit by Perry Cook):

- *bamboo*
- *barmodel*
- *cabasa*
- *crunch*
- *dripwater*
- *gogobel*
- *guiro*

- *mandol*
- *marimba*
- *moog*
- *sandpaper*
- *sekere*
- *shaker*
- *sleighbells*
- *stix*
- *tambourine*
- *vibes*
- *voice*

Also, see the *STK Opcodes* section for information on the STK opcodes.

Other models and emulations

- *lorenz*
- *planet*
- *prepiano*
- Fractal Number (Mandelbrot set) generator: *mandel*
- *chuap*
- *gendy*
- *gendyc*
- *gendyx*

A section on physical modeling using the waveguide principles can be found here: *Waveguide Physical Modeling*

## Phasors

The opcodes that generate a moving phase value:

- *ephasor*
- *phasor*
- *phasorbnk*
- *syncphasor*
- *phasor*

These opcodes are useful in combination with the *Table access* opcodes.

## Random (Noise) Generators

Opcodes that generate random numbers are:

- *betarnd*
- *bexprnd*
- *cauchy*
- *cuserrnd*
- *dusernd*
- *dust*
- *dust2*
- *exprand*
- *fractalnoise*
- *gauss*
- *gausstrig*
- *linrand*
- *noise*
- *pcauchy*
- *pinkish*
- *pinker*
- *poisson*
- *rand*
- *randh*
- *randi*
- *rnd31*
- *random*
- *randomh*
- *randomi*
- *trirand*
- *unirand*
- *urd*

- *weibull*
- *jitter*
- *jitter2*
- *trandom*

See *seed* which sets the global seed value for all x-class noise generators, as well as other opcodes that use a random call, such as *grain.rand*, *randh*, *randi*, *rnd(x)* and *birnd(x)* are not affected by seed.

See also functions which generate random numbers in the section *Random Functions*.

## Sample Playback

Opcodes that implement sample playback and looping are:

- *bbcutm*
- *bbcuts*
- *flooper*
- *flooper2*
- *loscil*
- *loscil3*
- *loscilx*
- *lphasor*
- *lposcil*
- *lposcil3*
- *lposcila*
- *lposcilsa*
- *lposcilsa2*
- *sndloop*
- *waveset*

See also the *Signal Input* section for other ways to input sound.

## Soundfonts

### Fluid Opcodes

The fluid family of opcodes wraps Peter Hannape's SoundFont 2 player, FluidSynth: *fluidEngine* for instantiating a FluidSynth engine, *fluidSetInterpMethod* for setting interpolation method for a channel in a FluidSynth engine, *fluidLoad* for loading SoundFonts, *fluidProgramSelect* for assigning presets from a

SoundFont to a FluidSynth engine's MIDI channel, *fluidNote* for playing a note on a FluidSynth engine's MIDI channel, *fluidCCi* for sending a controller message at i-time to a FluidSynth engine's MIDI channel, *fluidCCk* for sending a controller message at k-rate to a FluidSynth engine's MIDI channel. *fluidControl* for playing and controlling loaded Soundfonts (using 'raw' MIDI messages), *fluidOut* for receiving audio from a single FluidSynth engine, and *fluidAllOut* for receiving audio from all FluidSynth engines.

- *fluidAllOut*
- *fluidCCi*
- *fluidCCk*
- *fluidControl*
- *fluidEngine*
- *fluidLoad*
- *fluidNote*
- *fluidOut*
- *fluidProgramSelect*
- *fluidSetInterpMethod*

## "Old" Soundfont opcodes

These opcodes can also use soundfonts to generate sound. *sfplay* etc. were created for one purpose -- to use the samples in SoundFonts. The fluid opcodes were created for another purpose -- to use SoundFonts more or less the way they were designed to be used, i.e. using keyboard mappings, layers, internal processing, etc.

- *sflist*
- *sfinstr*
- *sfinstr3*
- *sfinstr3m*
- *sfinstrm*
- *sfload*
- *sfpassign*
- *sfplay*
- *sfplay3*
- *sfplay3m*
- *sfplaym*
- *sflooper*
- *sfplist*

- *sfpreset*

## Scanned Synthesis

Scanned synthesis is a variant of physical modeling, where a network of masses connected by springs is used to generate a dynamic waveform. The opcode *scanu* defines the mass/spring network and sets it in motion. The opcode *scans* follows a predefined path (trajectory) around the network and outputs the detected waveform. Several *scans* instances may follow different paths around the same network.

These are highly efficient mechanical modelling algorithms for both synthesis and sonic animation via algorithmic processing. They should run in real-time. Thus, the output is useful either directly as audio, or as controller values for other parameters.

The Csound implementation adds support for a scanning path or matrix. Essentially, this offers the possibility of reconnecting the masses in different orders, causing the signal to propagate quite differently. They do not necessarily need to be connected to their direct neighbors. Essentially, the matrix has the effect of “molding” this surface into a radically different shape.

To produce the matrices, the table format is straightforward. For example, for 4 masses we have the following grid describing the possible connections:

	1	2	3	4
1				
2				
3				
4				

Whenever two masses are connected, the point they define is 1. If two masses are not connected, then the point they define is 0. For example, a unidirectional string has the following connections: (1,2), (2,3), (3,4). If it is bidirectional, it also has (2,1), (3,2), (4,3)). For the unidirectional string, the matrix appears:

	1	2	3	4
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1
4	0	0	0	0

The above table format of the connection matrix is for conceptual convenience only. The actual values shown in the table are obtained by *scans* from an ASCII file using *GEN23*. The actual ASCII file is created from the table model row by row. Therefore the ASCII file for the example table shown above becomes:

```
0100001000010000
```

This matrix example is very small and simple. In practice, most scanned synthesis instruments will use many more masses than four, so their matrices will be much larger and more complex. See the example in the *scans* documentation.

Please note that the generated dynamic wavetables are very unstable. Certain values for masses, centering, and damping can cause the system to “blow up” and the most interesting sounds to emerge from your loudspeakers!

The supplement to this manual contains a tutorial on scanned synthesis. The tutorial, examples, and other information on scanned synthesis is available from the Scanned Synthesis page at cSounds.com.

Scanned synthesis developed by Bill Verplank, Max Mathews and Rob Shaw at Interval Research between 1998 and 2000.

Opcodes that implement scanned synthesis are:

- *scanhammer*
- *scans*
- *scantable*
- *scanu*
- *xscanmap*
- *xscans*
- *xscansmap*
- *xscanu*

## Table Access

The opcodes that access tables are:

- *oscill*
- *oscilli*
- *osciln*
- *oscilx*
- *table*
- *table3*
- *tablei*

Opcodes ending in 'i' implement linear interpolation and opcodes ending in '3' implement cubic interpolation.

The following opcodes implement fast table reading/writing without boundary checks:

- *tab*
- *tab\_i*
- *tabw*
- *tabw\_i*

See the sections *Table Queries*, *Read/Write Operations* and *Table Reading with Dynamic Selection* for other table operations.

**Note**

Although tables with a size which is not a power of two can be created using a negative size (see *f score statement*), some opcodes will not accept them.

## Wave Terrain Synthesis

The opcode that uses wave terrain synthesis is *wterrain*.

## Waveguide Physical Modeling

The opcodes that implement waveguide physical modeling are:

- *pluck*
- *repluck*
- *wgbow*
- *wgbowedbar*
- *wgbrass*
- *wgclar*
- *wgflute*
- *wgpluck*
- *wgpluck2*
- *wguide1*
- *wguide2*

---

# Signal Input and Output

## File Input and Output

The opcodes for file input and output are:

- File open/close: *fiopen* and *ficlose*.
- File output: *dumpk*, *dumpk2*, *dumpk3*, *dumpk4*, *fout*, *fouti*, *foutir* *foutk* and *hdf5write*
- File input: *readk*, *readk2*, *readk3*, *readk4*, *fin*, *fini* and *fink*
- Utilities for use with the *fout* opcodes: *clear*, *vincr*
- Printing to a file: *fprints* and *fprintks*

## Signal Input

The opcodes that receive audio signals are:

- Synchronous input: *in*, *in32*, *inch*, *inh*, *ino*, *inq*, *inrg*, *ins* and *inx*
- File streaming: *diskin*, *diskin2* *soundin* and *hdf5read*
- User defined channel input: *invalue*
- Streaming input: *soundin*
- Websocket input: *websocket*
- Direct to zak input: *inz*

See the section *Software Bus* for input and output through the API.

The *mp3in* allows reading of mp3 files, which are currently not supported by ordinary reading methods inside Csound.

## Signal Output

The opcodes that write audio signals are:

- Synchronous output: *out*, *out32*, *outc*, *outch*, *outh*, *outo*, *outrg*, *outq*, *outq1*, *outq2*, *outq3*, *outq4*, *outs*, *outs1*, *outs2* *outx* and *hdf5write*
- Streaming output: *soundout* and *soundouts*
- User defined channel output: *outvalue*
- Direct from zak output: *outz*
- Websocket output: *websocket*

The opcode *monitor* can be used for monitoring the complete output of *csound* (the output spout frame).

See the section *Software Bus* for input and output through the API.

## Software Bus

Csound implements a software bus for internal routing or routing to external software calling the Csound API.

The opcodes to use the software bus are:

- *chn\_k*
- *chn\_a*
- *chn\_S*
- *chnclear*
- *chnexport*
- *chnmix*
- *chnparams*

## Printing and Display

Opcodes for printing and displaying values are:

- *dispfft*
- *display*
- *flashtxt*
- *print*
- *printf*
- *printf\_i*
- *printk*
- *printk2*
- *printks*
- *prints*

## Sound File Queries

The opcodes that query information about files are:

- *filelen*
- *filenchnls*
- *filepeak*

- *filesr*
- *filevalid*

---

# Signal Modifiers

## Amplitude Modifiers and Dynamic processing

The opcodes that modify amplitude are:

- *balance*
- *compress*
- *clip*
- *dam*
- *gain*

The opcode *Odbfs* facilitates the use of amplitude by removing the need to use of explicit sample values.

## Convolution and Morphing

The opcodes that convolve and morph signals are:

- *convolve* also called *convle*
- *cross2*
- *dconv*
- *ftconv*
- *ftmorf*
- *pconvolve*

## Delay

### Fixed delays

- *delay*
- *delay1*
- *delayk*

### Delay Lines

- *delayr*
- *delayw*
- *deltap*

- *deltap3*
- *deltapi*
- *deltapn*
- *deltapx*
- *deltapxw*

## Variable delays

- *vdelay*
- *vdelay3*
- *vdelayx*
- *vdelayxs*
- *vdelayxq*
- *vdelayxw*
- *vdelayxwq*
- *vdelayxws*

## Multitap delays

- *multitap*

# Panning and Spatialization

## Amplitude spatialization

- *locsend*
- *locsig*
- *pan*
- *pan2*
- *space*
- *spdist*
- *spsend*

## 3D spatialization with simulation of room acoustics

- *spat3d*

- *spat3di*
- *spat3dt*

## Vector Base Amplitude Panning

- *vbap16*
- *vbap16move*
- *vbap4*
- *vbap4move*
- *vbap8*
- *vbap8move*
- *vbaplsinit*
- *vbapz*
- *vbapzmove*

## Binaural spatialization

- *hrtfer*
- *hrtfmove*
- *hrtfmove2*
- *hrtfstat*

## Ambisonics

- *bformdec*
- *bformenc*

## Reverberation

The opcodes one can use for reverberation are:

- *alpass*
- *babo*
- *comb*
- *freeverb*
- *nstedap*
- *nreverb* (also called *reverb2*)

- *reverb*
- *reverb**sc*
- *valpass*
- *vcomb*

## Sample Level Operators

The opcodes one may use to modify signals are:

- *a(k)*
- *denorm*
- *diff*
- *downsamp*
- *fold*
- *i(k)*
- *integ*
- *interp*
- *k(i)*
- *ntrpol*
- *samphold*
- *upsamp*
- *valet*
- *vaset*

## Signal Limiters

Opcodes that can be used to limit signals are:

- *limit*
- *mirror*
- *wrap*

## Special Effects

Opcodes that generate special effects are:

- *distort*

- *distort1*
- *exciter*
- *flanger*
- *harmon*
- *phaser1*
- *phaser2*

## Standard Filters

### Resonant Low-pass filters

- *areson*
- *lowpass2*
- *lowres*
- *lowresx*
- *lpf18*
- *moogvcf*
- *moogladder*
- *mvclpf1*
- *mvclpf2*
- *mvclpf3*
- *mvclpf4*
- *reson*
- *resonr*
- *resonx*
- *resony*
- *resonz*
- *rezy*
- *statevar*
- *svfilter*
- *tbvcf*
- *vlowres*

- *bqrez*

## Standard filters

- Hi-pass filters: *atone*, *atonex*, *mvchpf*
- Low-pass filters: *tone*, *tonex*
- Biquad filters: *biquad* and *biquada*.
- Butterworth filters: *butterbp*, *butterbr*, *butterhp*, *butterlp* (which are also called *butbp*, *butbr*, *buthp*, *butlp*)
- General filters: *clfilt*

## Zero-delay Feedback Filters (Virtual Analog)

- *zdf\_1pole*
- *zdf\_1pole\_mode*
- *zdf\_2pole*
- *zdf\_2pole\_mode*
- *zdf\_ladder*
- *diode\_ladder*
- *K35\_hpf*
- *K35\_lpf*

## Control signal filters

- *aresonk*
- *atonek*
- *lineto*
- *port*
- *portk*
- *resonk*
- *resonxk*
- *tilineto*
- *tonek*
- *sc\_lag*
- *sc\_lagud*

## Specialized Filters

### High pass filters

- *dcblock*
- *dcblock2*

### Parametric EQ

- *pareq*
- *rbjeq*
- *eqfil*

### Other filters

- *nlfilt*
- *filter2*
- *fofilter*
- *hilbert*
- *mode*
- *zfilter2*

## Waveguides

The opcodes that use waveguides to modify a signal are:

- *streson*
- *wguide1*
- *wguide2*

## Waveshaping and Phase Distortion

These opcodes can perform dynamic waveshaping or phaseshaping (a.k.a. phase distortion). They differ from traditional table-based methods of waveshaping by directly calculating the transfer function with one or more variable parameters for affecting the amount or results of the shaping. Most of these opcodes could be used on either an audio signal (for waveshaping) or a phasor (for phaseshaping) but tend to work best for one of these applications.

These opcodes are good for waveshaping:

- *chebyshevpoly*
- *clip*

- *distort*
- *distort1*
- *polynomial*
- *powershape*

These opcodes are good for phaseshaping:

- *pdclip*
- *pdhalf*
- *pdhalfy*

---

# Instrument Control

## Clock Control

The opcodes to start and stop internal clocks are:

- *clockoff*
- *clockon*

These clocks count CPU time. There are 32 independent clocks available. You can use the opcode *read-clock* to read current values of a clock. See *Time Reading* for other timing opcodes.

## Conditional Values

The opcodes for conditional values are ==, >=, >, <, <=, and !=.

## Duration Control Statements

The opcodes one can use to manipulate a note's duration are:

- *ihold*
- *turnoff*
- *turnoff2*
- *turnon*

For other realtime instrument control see *Real-time Performance Control* and *Instrument Invocation*.

## FLTK Widgets and GUI controllers

Widgets allow the design of a custom Graphical User Interface (GUI) to control an orchestra in real-time. They are derived from the open-source library FLTK (Fast Light Tool Kit). This library is one of the fastest graphic libraries available, supports OpenGL and should be source compatible with different platforms (Windows, Linux, Unix and Mac OS). The subset of FLTK implemented in Csound provides the following types of objects:

Containers

*FLTK Containers* are widgets that contain other widgets such as panels, windows, etc. Csound provides the following container objects:

- Panels
- Scroll areas
- Pack
- Tabs
- Groups

Valuators            The most useful objects are named *FLTK Valuators*. These objects allow the user to vary synthesis parameter values in real-time. Csound provides the following valuator objects:

- Sliders
- Knobs
- Rollers
- Text fields
- Joysticks
- Counters

Other widgets        There are *other FTLK widgets* that are not valuators nor containers:

- Buttons
- Button banks
- Labels
- Keyboard and Mouse sensing

Also there are some other opcodes useful to modify the *widget appearance*:

- Updating widget value.
- Setting primary and selection colors of a widget.
- Setting font type, size and color of widgets.
- Resizing a widget.
- Hiding and showing a widget.

There are also these *general opcodes* that allow the following actions:

- Running the widget thread: *FLrun*
- Loading snapshots containing the status of all valuators of an orchestra: *FLgetsnap* and *FLloadsnap*.
- Saving snapshots containing the status of all valuators of an orchestra: *FLsavesnap* and *FLsetsnap*
- Setting the snapshot group of a declared valuator: *FLsetSnapGroup*

Below is a simple example of Csound code to create a window. Notice that all opcodes are init-rate and must be called only once per session. The best way to use them is to place them in the header section of an orchestra, before any instrument. Even though placing them inside an instrument is not prohibited, unpredictable results can occur if that instrument is called more than once.

Each container is made up of a couple of opcodes: the first indicating the start of the container block and the last indicating the end of that container block. Some container blocks can be nested but they must not be crossed. After defining all containers, a widget thread must be run by using the special *FLrun* opcode that takes no arguments.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc      -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o linseg.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
;*****
sr=48000
kr=480
ksmps=100
nchnls=1

;*** It is recommended to put almost all GUI code in the
;*** header section of an orchestra

          FLpanel          "Panell",450,550 ;***** start of container
; some widgets should contained here
          FLpanelEnd      ;***** end of container

          FLrun           ;***** runs the widget thread, it is always required!
instr 1
;put some synthesis code here
endin
;*****
</CsInstruments>
<CsScore>
f 0 3600 ;dummy table for realtime input
e

</CsScore>
</CsoundSynthesizer>

```

The previous code simply creates a panel (an empty window because no widgets are defined inside the container).

The following example creates two panels and inserts a slider inside each of them:

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc      ; -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o linseg.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
;*****
sr=48000
kr=480
ksmps=100
nchnls=1

          FLpanel          "Panell",450,550,100,100 ;***** start of container
gk1,iha FLslider          "FLslider 1", 500, 1000, 0 ,1, -1, 300,15, 20,50
          FLpanelEnd      ;***** end of container

          FLpanel          "Panel2",450,550,100,100 ;***** start of container
gk2,ihb FLslider          "FLslider 2", 100, 200, 0 ,1, -1, 300,15, 20,50
          FLpanelEnd      ;***** end of container

          FLrun           ;***** runs the widget thread, it is always required!

```

```
instr 1
; gk1 and gk2 variables that contain the output of valuator
; widgets previously defined, can be used inside any instrument
printk2 gk1
printk2 gk2 ;print the values of the valuator whenever they change
endin
;*****
</CsInstruments>
<CsScore>
f 0 3600 ;dummy table for realtime input
e

</CsScore>
</CsoundSynthesizer>
```

All widget opcodes are init-rate opcodes, even if valuator output k-rate variables. This happens because an independent thread is run based on a callback mechanism. It consumes very few processing resources since there is no need of polling. (This differs from other MIDI based controller opcodes.) So you can use any number of windows and valuator without degrading the real-time performance.

## FLTK Containers

The opcodes for FLTK containers are:

- *FLgroup*
- *FLgroupEnd*
- *FLpack*
- *FLpackEnd*
- *FLpanel*
- *FLpanelEnd*
- *FLscroll*
- *FLscrollEnd*
- *FLtabs*
- *FLtabsEnd*

## FLTK Valuator

The opcodes for FLTK valuator are:

- *FLcount*
- *FLjoy*
- *FLknob*
- *FLroller*
- *FLslider*
- *FLtext*

## Other FLTK Widgets

Other FLTK widget opcodes are:

- *FLbox*
- *FLbutBank*
- *FLbutton*
- *FLexecButton*
- *FLkeyIn*
- *FLhvsBox*
- *FLhvsBoxSetValue*
- *FLmouse*
- *FLprintk*
- *FLprintk2*
- *FLslidBnk*
- *FLslidBnk2*
- *FLslidBnkGetHandle*
- *FLslidBnkSet*
- *FLslidBnk2Set*
- *FLslidBnk2Setk*
- *FLvalue*
- *FLvkeybd*
- *FLvslidBnk*
- *FLvslidBnk2*
- *FLxyin*

## Modifying FLTK Widget Appearance

The following opcodes modify FLTK widget appearance:

- *FLcolor*
- *FLcolor2*
- *FLhide*
- *FLlabel*

- *FLsetAlign*
- *FLsetBox*
- *FLsetColor*
- *FLsetColor2*
- *FLsetFont*
- *FLsetPosition*
- *FLsetSize*
- *FLsetText*
- *FLsetTextColor*
- *FLsetTextSize*
- *FLsetTextType*
- *FLsetVal\_i*
- *FLsetVal*
- *FLshow*

## General FLTK Widget-related Opcodes

The general FLTK widget-related opcodes are:

- *FLgetsnap*
- *FLloadsnap*
- *FLrun*
- *FLsavesnap*
- *FLsetsnap*
- *FLupdate*
- *FLsetSnapGroup*

## Instrument Invocation

The opcodes one can use to create score events from within an orchestra are:

- *event*
- *event\_i*
- *scoreline\_i*

- *scoreline*
- *schedule*
- *schedwhen*
- *schedkwhen*
- *schedkwhennamed*

The *mute* opcode can be used to mute/unmute instruments during a performance.

Instruments definitions can be removed using the *remove* opcode.

## Program Flow Control

The opcodes to manipulate which orchestra statements are executed are:

- *cgoto*
- *cigoto*
- *ckgoto*
- *cngoto*
- *elseif*
- *else*
- *endif*
- *goto*
- *if*
- *igoto*
- *kgoto*
- *tigoto*
- *timeout*

Opcodes to create looping constructions are:

- *loop\_ge*
- *loop\_gt*
- *loop\_le*
- *loop\_lt*
- *until*
- *while*



## Warning

Some of these opcodes work at i-rate even if they contain k- or a- rate comparisons. See the *Reinitialization* section.

# Real-time Performance Control

Opcodes that monitor and control real-time performance are:

- *active*
- *cpuprc*
- *maxalloc*
- *prealloc*
- *jacktransport*

The running *csound* process can be terminated using *exitnow*.

# Initialization and Reinitialization

Opcodes used for the initialization of variables:

- *init*
- *tival*
- *=*
- *passign*
- *pset*

The opcodes that can generate another initialization pass are:

- *reinit*
- *rigoto*
- *rireturn*

The opcode *p* can be used to find score p-fields at i- or k-rate.

*nstrnum* returns the instrument number for a named instrument.



## Note

Note that an instrument may modify the *p3* (duration) parameter at initialisation time. For example statements like

```
iattack = 0.02
irelease = 0.04
isustain = p3
p3 = iattack + isustain + irelease
```

are valid.

## Sensing and Control

### TCL/TK widgets

- *button*
- *checkbox*
- *control*
- *setctrl*

### Keyboard and mouse sensing

- *sensekey* (also called *sense*)
- *xyin*

### Envelope followers

- *follow*
- *follow2*
- *peak*
- *rms*

### Tempo and Pitch estimation

- *ptrack*
- *pitch*
- *pitchamdf*
- *tempest*

### Tempo and Sequencing

- *tempo*
- *miditempo*
- *tempoval*
- *seqtime*
- *seqtime2*
- *trigger*

- *trigseq*
- *timedseq*
- *changed*

## System

- *getcfg*

## Score control

- *rewindscore*
- *setscorepos*

## Stacks

Csound implements a global stack that can be accessed with the following opcodes:

- *stack*
- *pop*
- *push*
- *pop\_f*
- *push\_f*

## Sub-instrument Control

These opcodes let one define and use a sub-instrument:

- *subinstr*
- *subinstrinit*

See also the UDO and *Orchestra Macros* Macros section for similar functionality.

## Time Reading

Opcodes one can use to read time values are:

- *readclock*
- *rtclock*
- *timeinstk*
- *timeinsts*
- *times*

- *timek*

You can obtain the system date using:

- *date* - Returns the number seconds since 1 January 1970.
- *dates* - Returns as a string the date and time specified.

You can also set up counters using *clockoff* and *clockon*.

---

# Function Table Control

Refer to the *f score statement*, *ftgen*, *ftgentmp*, *ftgenonce* and the *GEN Routines* section for information on creating tables.

Tables can be removed from memory using the *ftfree* opcode.

Tables by default, require a size which is a power of two. However tables with any size can be generated by specifying the size as a negative number (see *f score statement*).



## Note

Not all opcodes accept tables whose size is not a power of two, as this may be a requirement for internal processing.

For information on table access, consult the section *Table Access*.

Tables for use with the *loscilx* opcode can be loaded using *sndload*.

## Table Queries

Opcodes the query tables for information are:

- For tables loaded from a sound file (using *GEN01*): *fchmls*, *ftcps*, *filen*, *ftlptim* and *ftsr*
- For any table: *nsamp*, *filen*, *tableng*

The opcode *tabsum* calculates the sum of values in a table.

## Read/Write Operations

Opcodes that read and write to a table are:

- *ftloadk*
- *ftload*
- *ftsavek*
- *ftsave*
- *tablecopy*
- *tablegpw*
- *tableicopy*
- *tableigpw*
- *tableimix*
- *tableiw*
- *tablemix*

- *tablera*
- *tablew*
- *tablewa*
- *tablewkt*
- *tabmorph*
- *tabmorpha*
- *tabmorphak*
- *tabmorphi*
- *tabrec*
- *tabplay*
- *fmorf*

Table values can be accessed within expressions using the *tb* family of opcodes.

Many oscillators are in fact specialized table readers. See the *Basic oscillators* section.

## Table Reading with Dynamic Selection

Opcodes that let one dynamically (at k-rate) select tables are:

- *tableikt*
- *tablekt*
- *tablexkt*

---

# Mathematical Operations

## Amplitude Converters

Opcodes to convert between different amplitude measurements are:

- *ampdb*
- *ampdbfs*
- *db*
- *dbamp*
- *dbfsamp*

Use *rms* to find the rms value of a signal. See also *Odbfs* for another way to handle amplitudes in *csound*.

## Arithmetic and Logic Operations

Opcodes that perform arithmetic and logic operations are -, +, &&, //, \*, /, ^, and %.

See the *Conditional Values* section and the *if* family of opcodes for usage of logical operators.

## Comparators and Accumulators

The following opcodes perform comparisons between signals at a-rate or k-rate, find maxima or minima, or accumulate the results of several computations or comparisons:

- *max*
- *max\_k*
- *maxabs*
- *maxabsaccum*
- *maxaccum*
- *min*
- *minabs*
- *minabsaccum*
- *minaccum*
- *vincr*
- *clear*
- *cmp*

## Mathematical Functions

Opcodes that perform mathematical functions are:

- *abs*
- *ceil*
- *exp*
- *floor*
- *frac*
- *int*
- *log*
- *log10*
- *logbtwo*
- *pow*
- *powershape*
- *powoftwo*
- *round*
- *sqrt*

## Opcode Equivalents of Functions

Opcodes that perform the equivalent of mathematical functions are:

- *chebyshevpoly*
- *divz*
- *mac*
- *maca*
- *polynomial*
- *pow*
- *product*
- *sum*
- *taninv2*

## Random Functions

Opcodes that perform random functions are:

- *birnd*
- *rnd*

See the section *Random (Noise) Generators* for opcodes that generate random signals.

## Trigonometric Functions

Opcodes that perform trigonometric functions are:

- *cos*, *cosh* and *cosinv*
- *sin*, *sinh* and *sininv*
- *tan*, *tanh*, *taninv*, and *taninv2*.

# Linear Algebra Opcodes

Linear Algebra Opcodes — Scalar, vector, and matrix arithmetic on real and complex values.

## Description

These opcodes implement many linear algebra operations, from scalar, vector, and matrix arithmetic up to and including QR based eigenvalue decompositions. The opcodes are designed for digital signal processing, and of course other mathematical operations, in the Csound orchestra language.

The numerical implementation uses the gmm++ library from *home.gna.org/getfem/gmm\_intro* [[http://home.gna.org/getfem/gmm\\_intro](http://home.gna.org/getfem/gmm_intro)].



### Warning

For applications with f-sig variables, array arithmetic must be performed only when the f-sig is "current," because f-rate is some fraction of k-rate; currency can be determined with the `la_k_current_f` opcode.

For applications using assignments between real vectors and a-rate variables, array arithmetic must be performed only when the vectors are "current", because the size of the vector may be some integral multiple of ksmps; currency can be determined by means of the `la_k_current_vr` opcode.

**Table 4. Linear Algebra Data Types**

Mathematical Type	Code	Corresponding Csound Type or Types
real scalar	r	i-rate or k-rate variable
complex scalar	c	pair of i-rate or k-rate variables, e.g. "kr, ki"
real vector	vr	i-rate variable holding address of array
real vector	a	a-rate variable
real vector	t	function table number
complex vector	vc	i-rate variable holding address of array
complex vector	f	fsig variable
real matrix	mr	i-rate variable holding address of array
complex matrix	mc	i-rate variable holding address of array

All arrays are 0-based; the first index iterates rows to give columns, the second index iterates columns to give elements.

All arrays are general and dense; banded, Hermitian, symmetric and sparse routines are not implemented.

An array can be of type code vr, vc, mr, or mc and is stored in an i-rate object. In orchestra code, an array is passed as a MYFLT i-rate variable that contains the address of the array object, which is actually stored in the allocator opcode instance. Although array variables are i-rate, of course their values and even shapes may change at i-rate or k-rate.

All operands must be pre-allocated; except for the creation opcodes, no opcode ever allocates any arrays. This is true even if the array appears on the left-hand side of an opcode! However, some operations may reshape arrays to hold results.

Arrays are automatically deallocated when their instrument is deallocated.

Not only for more efficient performance, but also to make it easier to remember opcode names, the performance rate, output value types, operation names, and input value types are deterministically encoded into the opcode name:

1. "la" for "linear algebra opcode family".
2. "i" or "k" for performance rate.
3. Type code(s) (see above table) for output value(s), but only if the type is not implicit from the input values.
4. Operation name: common mathematical name (preferred) or abbreviation.
5. Type code(s) for input values, if not implicit.

For additional details, see the gmm++ documentation at <http://download.gna.org/getfem/doc/gm-muser.pdf>.

## Syntax

### Array Creation

`ivr`                                    `la_i_vr_create`                    `irows`

Create a real vector with `irows` rows.

`ivc`                                    `la_i_vc_create`                    `irows`

Create a complex vector with `irows` rows.

`imr`                                    `la_i_mr_create`                    `irows, icolumns [, odiagonal]`

Create a real matrix with `irows` rows and `icolumns` columns, with an optional value on the diagonal.

`imc`                                    `la_i_mc_create`                    `irows, icolumns [, odiagonal_r, odiagonal_i]`

Create a complex matrix with `irows` rows and `icolumns` columns, with an optional complex value on the diagonal.

### Array Introspection

`irows`                                    `la_i_size_vr`                    `ivr`

Return the number of rows in real vector `ivr`.

`irows`                                    `la_i_size_vc`                    `ivc`

Return the number of rows in complex vector `ivc`.

`irows, icolumns`                    `la_i_size_mr`                    `imr`

Return the number of rows and columns in real matrix `imr`.

`irows, icolumns`                    `la_i_size_mc`                    `imc`

Return the number of rows and columns in complex matrix `imc`.

`kfiscurrent`                        `la_k_current_f`                    `fsg`

Return 1 if `fsg` is current, that is, if the value of `fsg` will change on the next `kperiod`.

`kvriscurrent`                        `la_k_current_vr`                    `ivr`

Return 1 if the real vector `ivr` is current, that is, if Csound's current audio sample frame stands at index 0 of the vector.

`la_i_print_vr`                        `ivr`

Print the value of real vector `ivr`.

`la_i_print_vc`                        `ivc`

Print the value of complex vector `ivc`.

`la_i_print_mr`                        `imr`

Print the value of real matrix `imr`.

`la_i_print_mc`                        `imc`

Print the value of complex matrix `imc`.

## Array Assignment and Conversion

`ivr`                                    `la_i_assign_vr`                    `ivr`

Assign the value of the real vector on the right-hand side to the real vector on the left-hand side, at `i-rate`.

`ivr`                                    `la_k_assign_vr`                    `ivr`

Assign the value of the real vector on the right-hand side to the real vector on the left-hand side, at `k-rate`.

`ivc`                                    `la_i_assign_vc`                    `ivc`

`ivc`                                    `la_k_assign_vc`                    `ivr`

`imr`                                    `la_i_assign_mr`                    `imr`

`imr`                                    `la_k_assign_mr`                    `imr`

`imc`                                    `la_i_assign_mc`                    `imc`

`imc`                                    `la_k_assign_mc`                    `imr`



### Warning

Assignments to vectors from tables or `fsgs` may resize the vectors.

Assignments to vectors from `a-rate` variables, or to `a-rate` variables from vectors, will be performed incrementally, one chunk of `ksmps` elements per `kperiod`. Therefore, array arithmetic on such vectors should only be performed when the vectors are current, as determined by the `la_k_current_vr` opcode.

ivr	la_k_assign_a	asig
ivr	la_i_assign_t	itablenumber
ivr	la_k_assign_t	itablenumber
ivc	la_k_assign_f	fsig
asig	la_k_a_assign	ivr
itablenum	la_i_t_assign	ivr
itablenum	la_k_t_assign	ivr
fsig	la_k_f_assign	ivc

## Fill Arrays with Random Elements

ivr	la_i_random_vr	[ifill_fraction]
ivr	la_k_random_vr	[kfill_fraction]
ivc	la_i_random_vc	[ifill_fraction]
ivc	la_k_random_vc	[kfill_fraction]
imr	la_i_random_mr	[ifill_fraction]
imr	la_k_random_mr	[kfill_fraction]
imc	la_i_random_mc	[ifill_fraction]
imc	la_k_random_mc	[kfill_fraction]

## Array Element Access

ivr	la_i_vr_set	irow, ivalue
kvr	la_k_vr_set	krow, kvalue
ivc	la_i_vc_set	irow, ivalue_r, ivalue_i
kvc	la_k_vc_set	krow, kvalue_r, kvalue_i
imr	la_i_mr_set	irow, icolumn, ivalue
kmr	la_k_mr_set	krow, kcolumn, ivalue
imc	la_i_mc_set	irow, icolumn, ivalue_r, ivalue_i
kmc	la_k_mc_set	krow, kcolumn, kvalue_r, kvalue_i
ivalue	la_i_get_vr	ivr, irow
kvalue	la_k_get_vr	ivr, krow
ivalue_r, ivalue_i	la_i_get_vc	ivc, irow
kvalue_r, kvalue_i	la_k_get_vc	ivc, krow
ivalue	la_i_get_mr	imr, irow, icolumn
kvalue	la_k_get_mr	imr, krow, kcolumn
ivalue_r, ivalue_i	la_i_get_mc	imc, irow, icolumn
kvalue_r, kvalue_i	la_k_get_mc	imc, krow, kcolumn

## Single Array Operations

imr	la_i_transpose_mr	imr
-----	-------------------	-----

imr	la_k_transpose_mr	imr
imc	la_i_transpose_mc	imc
imc	la_k_transpose_mc	imc
ivr	la_i_conjugate_vr	ivr
ivr	la_k_conjugate_vr	ivr
ivc	la_i_conjugate_vc	ivc
ivc	la_k_conjugate_vc	ivc
imr	la_i_conjugate_mr	imr
imr	la_k_conjugate_mr	imr
imc	la_i_conjugate_mc	imc
imc	la_k_conjugate_mc	imc

## Scalar Operations

ir	la_i_norm1_vr	ivr
kr	la_k_norm1_vr	ivc
ir	la_i_norm1_vc	ivc
kr	la_k_norm1_vc	ivc
ir	la_i_norm1_mr	imr
kr	la_k_norm1_mr	imr
ir	la_i_norm1_mc	imc
kr	la_k_norm1_mc	imc
ir	la_i_norm_euclid_vr	ivr
kr	la_k_norm_euclid_vr	ivr
ir	la_i_norm_euclid_vc	ivc
kr	la_k_norm_euclid_vc	ivc
ir	la_i_norm_euclid_mr	mvr
kr	la_k_norm_euclid_mr	mvr
ir	la_i_norm_euclid_mc	mvc
kr	la_k_norm_euclid_mc	mvc
ir	la_i_distance_vr	ivr
kr	la_k_distance_vr	ivr
ir	la_i_distance_vc	ivc
kr	la_k_distance_vc	ivc
ir	la_i_norm_max	imr
kr	la_k_norm_max	imc
ir	la_i_norm_max	imr
kr	la_k_norm_max	imc
ir	la_i_norm_inf_vr	ivr

kr	la_k_norm_inf_vr	ivr
ir	la_i_norm_inf_vc	ivc
kr	la_k_norm_inf_vc	ivc
ir	la_i_norm_inf_mr	imr
kr	la_k_norm_inf_mr	imr
ir	la_i_norm_inf_mc	imc
kr	la_k_norm_inf_mc	imc
ir	la_i_trace_mr	imr
kr	la_k_trace_mr	imr
ir, ii	la_i_trace_mc	imc
kr, ki	la_k_trace_mc	imc
ir	la_i_lu_det	imr
kr	la_k_lu_det	imr
ir	la_i_lu_det	imc
kr	la_k_lu_det	imc

## Elementwise Array-Array Operations

ivr	la_i_add_vr	ivr_a, ivr_b
ivc	la_k_add_vc	ivc_a, ivc_b
imr	la_i_add_mr	imr_a, imr_b
imc	la_k_add_mc	imc_a, imc_b
ivr	la_i_subtract_vr	ivr_a, ivr_b
ivc	la_k_subtract_vc	ivc_a, ivc_b
imr	la_i_subtract_mr	imr_a, imr_b
imc	la_k_subtract_mc	imc_a, imc_b
ivr	la_i_multiply_vr	ivr_a, ivr_b
ivc	la_k_multiply_vc	ivc_a, ivc_b
imr	la_i_multiply_mr	imr_a, imr_b
imc	la_k_multiply_mc	imc_a, imc_b
ivr	la_i_divide_vr	ivr_a, ivr_b
ivc	la_k_divide_vc	ivc_a, ivc_b
imr	la_i_divide_mr	imr_a, imr_b
imc	la_k_divide_mc	imc_a, imc_b

## Inner Products

ir	la_i_dot_vr	ivr_a, ivr_b
kr	la_k_dot_vr	ivr_a, ivr_b
ir, ii	la_i_dot_vc	ivc_a, ivc_b

kr, ki	<code>la_k_dot_vc</code>	ivc_a, ivc_b
imr	<code>la_i_dot_mr</code>	imr_a, imr_b
imr	<code>la_k_dot_mr</code>	imr_a, imr_b
imc	<code>la_i_dot_mc</code>	imc_a, imc_b
imc	<code>la_k_dot_mc</code>	imc_a, imc_b
ivr	<code>la_i_dot_mr_vr</code>	imr_a, ivr_b
ivr	<code>la_k_dot_mr_vr</code>	imr_a, ivr_b
ivc	<code>la_i_dot_mc_vc</code>	imc_a, ivc_b
ivc	<code>la_k_dot_mc_vc</code>	imc_a, ivc_b

## Matrix Inversion

imr, icondition	<code>la_i_invert_mr</code>	imr
imr, kcondition	<code>la_k_invert_mr</code>	imr
imc, icondition	<code>la_i_invert_mc</code>	imc
imc, kcondition	<code>la_k_invert_mc</code>	imc

## Matrix Decompositions and Solvers

ivr	<code>la_i_upper_solve_mr</code>	imr [, j_1_diagonal]
ivr	<code>la_k_upper_solve_mr</code>	imr [, j_1_diagonal]
ivc	<code>la_i_upper_solve_mc</code>	imc [, j_1_diagonal]
ivc	<code>la_k_upper_solve_mc</code>	imc [, j_1_diagonal]
ivr	<code>la_i_lower_solve_mr</code>	imr [, j_1_diagonal]
ivr	<code>la_k_lower_solve_mr</code>	imr [, j_1_diagonal]
ivc	<code>la_i_lower_solve_mc</code>	imc [, j_1_diagonal]
ivc	<code>la_k_lower_solve_mc</code>	imc [, j_1_diagonal]
imr, ivr_pivot, isize	<code>la_i_lu_factor_mr</code>	imr
imr, ivr_pivot, ksize	<code>la_k_lu_factor_mr</code>	imr
imc, ivr_pivot, isize	<code>la_i_lu_factor_mc</code>	imc
imc, ivr_pivot, ksize	<code>la_k_lu_factor_mc</code>	imc
ivr_x	<code>la_i_lu_solve_mr</code>	imr, ivr_b
ivr_x	<code>la_k_lu_solve_mr</code>	imr, ivr_b
ivc_x	<code>la_i_lu_solve_mc</code>	imc, ivc_b
ivc_x	<code>la_k_lu_solve_mc</code>	imc, ivc_b
imr_q, imr_r	<code>la_i_qr_factor_mr</code>	imr
imr_q, imr_r	<code>la_k_qr_factor_mr</code>	imr
imc_q, imc_r	<code>la_i_qr_factor_mc</code>	imc
imc_q, imc_r	<code>la_k_qr_factor_mc</code>	imc
ivr_eig_vals	<code>la_i_qr_eigen_mr</code>	imr, i_tolerance

<code>ivr_eig_vals</code>	<code>la_k_qr_eigen_mr</code>	<code>imr, k_tolerance</code>
<code>ivr_eig_vals</code>	<code>la_i_qr_eigen_mc</code>	<code>imc, i_tolerance</code>
<code>ivr_eig_vals</code>	<code>la_k_qr_eigen_mc</code>	<code>imc, k_tolerance</code>



### Warning

Matrix must be Hermitian in order to compute eigenvectors.

<code>ivr_eig_vals, imr_eig_vecs</code>	<code>la_i_qr_sym_eigen_mr</code>	<code>imr, i_tolerance</code>
<code>ivr_eig_vals, imr_eig_vecs</code>	<code>la_k_qr_sym_eigen_mr</code>	<code>imr, k_tolerance</code>
<code>ivc_eig_vals, imc_eig_vecs</code>	<code>la_i_qr_sym_eigen_mc</code>	<code>imc, i_tolerance</code>
<code>ivc_eig_vals, imc_eig_vecs</code>	<code>la_k_qr_sym_eigen_mc</code>	<code>imc, k_tolerance</code>

## Credits

Michael Gogins

New in Csound version 5.09

# Array Opcodes

Array Opcodes

## Variable Name

An array must be created (via *init* or *fillarray*) as `kMyName` plus ending brackets. The brackets determine the dimensions of the array. So,

```
kArr[] init 10
```

creates a one-dimensional array of length 10, whereas

```
kArr[][] init 10, 10
```

creates a two-dimensional array with 10 rows and 10 columns.

After the initialization of the array, referring to the array as a whole is done **without** any brackets. Brackets are only used if an element is indexed:

```
kArr[] init 10           ;with brackets because of initialization  
kLen   = lenarray(kArr) ;without brackets  
kFirstEl = kArr[0]      ;indexing with brackets
```

The same syntax is used for a simple copy via the '=' operator:

```
kArr1[] fillarray 1, 2, 3, 4, 5  
kArr2[] = kArr1           ;creates kArr2 as copy of kArr1
```

## k-rate

Note that most array operations are currently k-rate only. So like any other k-rate opcode, an operation on arrays will be automatically repeated every k-cycle. For instance, this code will repeat re-writing the array with different random values every k-cycle, as long as the instrument runs:

```
kArr[] init 10  
kIndx = 0  
until kIndx == lenarray(kArr) do  
  kArr[kIndx] rnd31 10, 0  
  kIndx += 1  
od
```

If you want to avoid this, you must organize it in one of the usual ways, for instance by using a trigger:

```
kArr[] init 10  
kTrig metro 1  
if kTrig == 1 then ;do the following once a second  
  kIndx = 0  
  until kIndx == lenarray(kArr) do  
    kArr[kIndx] rnd31 10, 0  
    kIndx += 1  
  od  
endif
```

## Creation/Initialization

The usual way to create an array is with `init`:

```
kArr[]  init 10      ;creates one-dimensional array with length 10
kArr[][] init 10, 10 ;creates two-dimensional array
```

A one-dimensional array can also be created and filled with distinct values by the opcode `fillarray`. This line creates a vector with length 4 and puts in the numbers [1, 2, 3, 4]:

```
kArr[] fillarray 1, 2, 3, 4
```

## Length

The function `lenarray(kArr)` reports the length of an array. See example for function `lenarray`.

## Copy Arrays to/from Tables

```
copyf2array kArr, kfn
```

copies data from an ftable to a vector.

```
copya2ftab kArr, kfn
```

copies data from a vector to an function table.

See examples for opcodes `copyf2array` and `copya2ftab`.

## Array Operations: Math

### +, -, \*, / on a Number

If the four basic math operators are used between an array and a scalar (number), the operation is applied to each element. The safest way to do this is to store the result in a new array:

```
kArr1[] fillarray 1, 2, 3
kArr2[] = kArr1 + 10      ;(kArr2 is now [11, 12, 13])
```

Here is an example of array/scalar operations. It uses the file `array_scalar_math.csd` [examples/array\_scalar\_math.csd].

### Example 3. Example of array operations

```
<CsoundSynthesizer>
<CsOptions>
-n -m128
</CsOptions>
<CsInstruments>

instr 1

;create array and fill with numbers 1..10
kArr1[] fillarray 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

;print content
```

```

        printf "%s", 1, "\nInitial content:\n"
kndx = 0
    until kndx == lenarray(kArr1) do
        printf "kArr[%d] = %f\n", kndx+1, kndx, kArr1[kndx]
kndx += 1
    od

;add 10
kArr2[] = kArr1 + 10

;print content
        printf "%s", 1, "\nAfter adding 10:\n"
kndx = 0
    until kndx == lenarray(kArr2) do
        printf "kArr[%d] = %f\n", kndx+1, kndx, kArr2[kndx]
kndx += 1
    od

;subtract 5
kArr3[] = kArr2 - 5

;print content
        printf "%s", 1, "\nAfter subtracting 5:\n"
kndx = 0
    until kndx == lenarray(kArr3) do
        printf "kArr[%d] = %f\n", kndx+1, kndx, kArr3[kndx]
kndx += 1
    od

;multiply by -1.5
kArr4[] = kArr3 * -1.5

;print content
        printf "%s", 1, "\nAfter multiplying by -1.5:\n"
kndx = 0
    until kndx == lenarray(kArr4) do
        printf "kArr[%d] = %f\n", kndx+1, kndx, kArr4[kndx]
kndx += 1
    od

;divide by -3/2
kArr5[] = kArr4 / -(3/2)

;print content
        printf "%s", 1, "\nAfter dividing by -3/2:\n"
kndx = 0
    until kndx == lenarray(kArr5) do
        printf "kArr[%d] = %f\n", kndx+1, kndx, kArr5[kndx]
kndx += 1
    od

;turnoff
    turnoff
    endin

</CsInstruments>
<CsScore>
i 1 0 .1
</CsScore>
</CsSoundSynthesizer>

```

## + , - , \* , / on a Second Array

If the four basic math operators are used between two arrays, the operation is applied element by element. The result can be straightforward stored in a new array:

```
kArr1[] fillarray 1, 2, 3
kArr2[] fillarray 10, 20, 30
kArr3[] = kArr1 + kArr2 ;(kArr3 is now [11, 22, 33])
```

Here is an example of array operations. It uses the file *array\_array\_math.csd* [examples/array\_array\_math.csd].

#### Example 4. Example of array operations

```
<CsoundSynthesizer>
<CsOptions>
-n -m128
</CsOptions>
<CsInstruments>

instr 1

;create array and fill with numbers 1..10 resp .1..1
kArr1[] fillarray 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
kArr2[] fillarray 1, 2, 3, 5, 8, 13, 21, 34, 55, 89

;print contents
printf "%s", 1, "\nkArr1:\n"
kndx = 0
until kndx == lenarray(kArr1) do
printf "kArr1[%d] = %f\n", kndx+1, kndx, kArr1[kndx]
kndx += 1
od

printf "%s", 1, "\nkArr2:\n"
kndx = 0
until kndx == lenarray(kArr2) do
printf "kArr2[%d] = %f\n", kndx+1, kndx, kArr2[kndx]
kndx += 1
od

;add arrays
kArr3[] = kArr1 + kArr2

;print content
printf "%s", 1, "\nkArr1 + kArr2:\n"
kndx = 0
until kndx == lenarray(kArr3) do
printf "kArr3[%d] = %f\n", kndx+1, kndx, kArr3[kndx]
kndx += 1
od

;subtract arrays
kArr4[] = kArr1 - kArr2

;print content
printf "%s", 1, "\nkArr1 - kArr2:\n"
kndx = 0
until kndx == lenarray(kArr4) do
printf "kArr4[%d] = %f\n", kndx+1, kndx, kArr4[kndx]
kndx += 1
od

;multiply arrays
kArr5[] = kArr1 * kArr2

;print content
printf "%s", 1, "\nkArr1 * kArr2:\n"
kndx = 0
until kndx == lenarray(kArr5) do
```

```

        printf "kArr5[%d] = %f\n", kndx+1, kndx, kArr5[kndx]
kndx += 1
    od

;divide arrays
kArr6[] =      kArr1 / kArr2

;print content
        printf "%s", 1, "\nkArr1 / kArr2:\n"
kndx    =      0
    until kndx == lenarray(kArr6) do
        printf "kArr5[%d] = %f\n", kndx+1, kndx, kArr6[kndx]
kndx += 1
    od

;turnoff
        turnoff

    endin

</CsInstruments>
<CsScore>
i 1 0 .1
</CsScore>
</CsoundSynthesizer>

```

## Map a Function to an Array

```
kArrRes maparray kArrSrc, "fun"
```

maps the k-rate 1-arg function in the string to every element of the vector.

Possible functions are for instance *abs*, *ceil*, *exp*, *floor*, *frac*, *int*, *log*, *log10*, *round*, *sqrt*. This is a simple example:

```

kArrSrc[] fillarray 1, 2, 3, 4, 5
kArrRes[] init      5
kArrRes  maparray  kArrSrc, "sqrt"

```

See example for opcode *maparray*.

## Array Operations: min, max, sum, scale, slice

### Minimum and Maximum

```
kMin [,kMinIndx] minarray kArr
```

returns the smallest value in an array, and optionally its index.

```
kMax [,kMaxIndx] maxarray kArr
```

returns the largest value in an array, and optionally its index. See examples for opcodes *minarray* and *maxarray*.

### Sum

```
kSum sumarray kArr
```

returns the sum of all values in *kArr*. See example for opcode *sumarray*.

## Scale

```
scalearray kArr, kMin, kMax
```

scales all values in *kArr* between *kMin* and *kMax*.

```
kArr[] fillarray 1, 3, 9, 5, 6
      scalearray kArr, 1, 3
```

changes *kArr* to [1, 1.5, 3, 2, 2.25]. See example for opcode *scalearray*.

## Slice

```
slicearray kArr, iStart, iEnd
```

returns a slice of *kArr* from index *iStart* to index *iEnd* (included).

The array for receiving the slice must have been created in advance:

```
kArr[] fillarray 1, 2, 3, 4, 5, 6, 7, 8, 9
kArr1[] init      5
kArr2[] init      4
kArr1 slicearray kArr, 0, 4      ;[1, 2, 3, 4, 5]
kArr2 slicearray kArr, 5, 8      ;[6, 7, 8, 9]
```

See example for opcode *slicearray*.

## Arrays in UDOs

The dimension of an input array must be declared in two places:

- as *k[]* or *k[][]* in the type input list
- as *kName[]*, *kName[][]* etc in the *xin* list.

For instance :

```
opcode FirstEl, k, k[]
;returns the first element of vector kArr
kArr[] xin
      xout kArr[0]
endop
```

Here is an example of an array in an UDO. It uses the file *array\_udo.csd* [examples/array\_udo.csd].

### Example 5. Example of an array in an UDO

```
<CsoundSynthesizer>
<CsoundOptions>
-nm128
```

```

</CsOptions>
<CsInstruments>

  opcode FirstEl, k, k[]
  ;returns the first element of vector kArr
kArr[] xin
xout kArr[0]
endop

  instr 1
kArr[] fillarray 6, 3, 9, 5, 1
kFirst FirstEl kArr
printf "kFirst = %d\n", 1, kFirst
turnoff
endin

</CsInstruments>
<CsScore>
i 1 0 .1
</CsScore>
</CsoundSynthesizer>

```

Note that if an opcode (for example `inrg`), alters arguments on its right hand argument list, an array index should not be used there. Unlike a normal variable, the array won't be changed by the opcode.

## Credits

This manual page has been written by Joachim Heintz.  
 July 2013

New in Csound 6.00

---

# Pitch Converters

## Functions

Opcodes that provide common pitch functions are:

- *cent*
- *cpsmidinn*
- *cpsoct*
- *cpspch*
- *octave*
- *octcps*
- *octmidinn*
- *octpch*
- *pchmidinn*
- *pchoct*
- *semitone*

## Tuning Opcodes

Opcodes that provide tuning functions are:

- *cps2pch*
- *cpsxpch*
- *cpstun*
- *cpstuni*

---

# Real-time MIDI Support

Csound supports realtime MIDI input and output, as well as input from MIDI files. Realtime MIDI input is activated using the `-M` (or `--midi-device=DEVICE`) command line flag. You must specify the device number or name after the `-M`. For example to use device number 2, you would use something like:

```
csound -M2 myrtmidi.csd
```

You can find out the available devices by using an out of range device:

```
csound -M99 myrtmidi.csd
```



## Note

This will only work if the MIDI module can be accessed by device number. For also, you must first find the device name using:

```
cat /proc/asound/cards
```

You must then use something like:

```
csound --rtmidi=alsa -M hw:3 myrtmidi.csd
```

Realtime MIDI output is activated using `-Q`, using device number or names as shown above.

You can also load a MIDI file using the `-F` or `--midifile=FILE` command line flag. The MIDI file is read in realtime, and behaves as if it was being performed or received in realtime. So the csound program is not aware if MIDI input comes from a MIDI file or directly from a MIDI interface.

Once realtime MIDI input and/or output has been activated, opcodes like *MIDI Input* and *MIDI Output* will have effect.

When MIDI input is enabled (with `-M` or `-F`), each incoming *noteon* message will generate a note event for an instrument which has the same number as the channel of the event (This means that MIDI controlled instruments are polyphonic by default, since each note will generate a new instance of the instrument.) If you have 1 instrument only, Csound works in omni mode, ie. it responds to all channels into that single instrument. If you have more than one instrument and instrs 1 - 16, then by default instr 1 -> chn 1, instr 2 -> chn 2, unless you alter the mapping (see *massign* and *pgmassign* to change this behavior). If you have more than one instrument, but instr N in between 1 - 16 is missing, then chn N will be routed by default to the lowest order instrument.

See the *MIDI/Score Interoperability* opcodes for information on designing instruments which can be used from the score or driven by MIDI.

There are several realtime MIDI modules available, you must use the `--rtmidi` flag (See `--rtmidi`), to specify the module. The default module is `portmidi` which provides adequate MIDI I/O on all platforms, however for improved performance and reliability some platform specific modules are also provided.

Currently the midi modules available are:

- *alsa* - To use the ALSA midi system (Linux only)
- *jack* - To use the Jack midi system
- *winnme* - To use the windows MME system (Windows only)
- *portmidi* - To use the portmidi system (all platforms). This is the default setting.

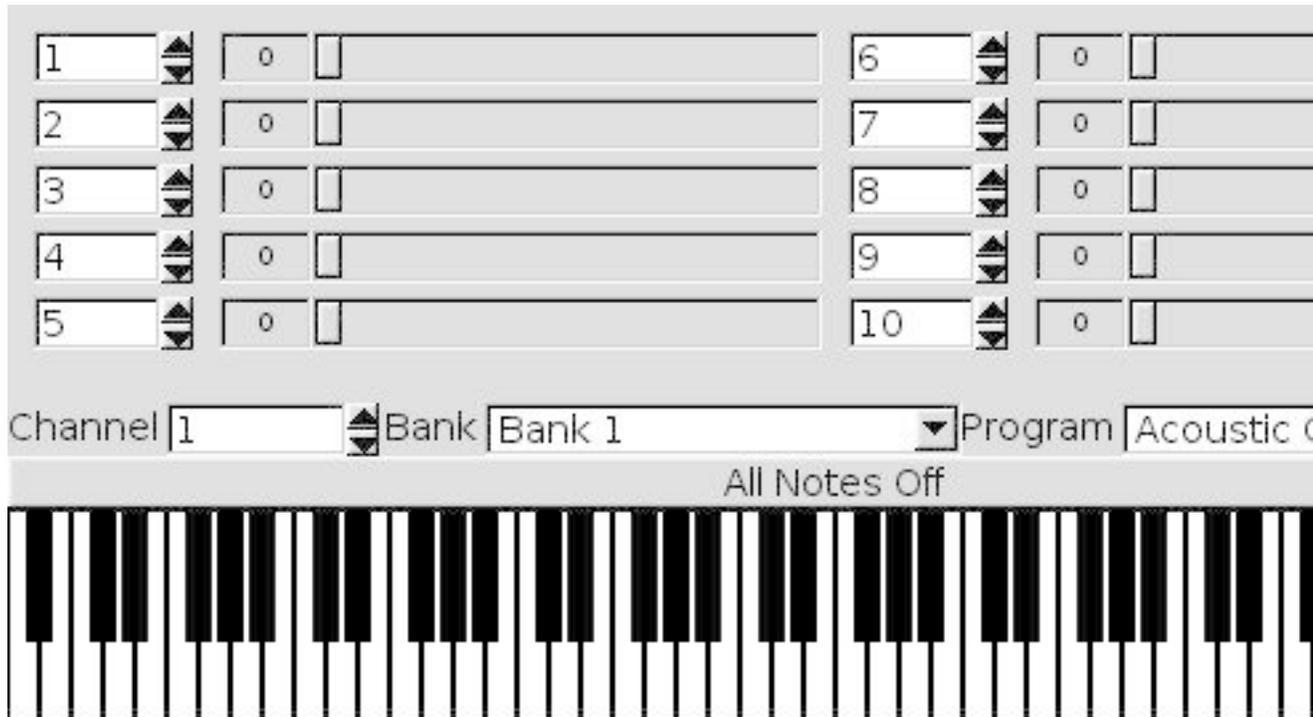
- *virtual* - To use a virtual graphical keyboard (See below) as MIDI input (all platforms)



### Tip

When csound runs, it will process the score and then quit. If there are no events in the score, Csound will exit immediately. If you want to use only MIDI events instead of score events, you need to tell Csound to run for a certain amount of time. This can be done with a dummy *f-statement* like "f 0 3600".

## Virtual MIDI Keyboard



Virtual MIDI keyboard.

The virtual MIDI keyboard module (activated using `-+rtmidi=virtual` on the command line flags) provides a way of sending realtime MIDI information to Csound without the need of a MIDI device. It can send note information, control changes, bank and program changes on a specified channel. The MIDI information from the virtual keyboard is processed by Csound in exactly the same way as MIDI information that comes from the other MIDI drivers, so if your Csound orchestra is designed to work with hardware MIDI devices, this will also work.

For the device flag (`-M`), the virtual keyboard uses this to take in the name of a keyboard mapping files. Like all MIDI drivers, a device must be given to activate the driver. If you would like to just use the default settings of the keyboard, simply passing in 0 (i.e. `-M0`) and the virtual keyboard will use its default settings. If instead of the 0 a name of a file is given, the keyboard will attempt to load the file as a keyboard mapping. If the file could not be opened or read correctly, the default settings will be used.

Keyboard Mapping files allow the user to customize the name and number of banks as well as the name and number of programs per bank. The following example keyboard mapping (named `keyboard.map`) has inline comments on the file format. This file is also available with the Csound source distribution in the `InOut/virtual_keyboard` folder.

```

# Custom Keyboard Map for Virtual Keyboard
# Steven Yi
#
# USAGE
#
# When using the Virtual Keyboard, you can supply a filename for a mapping
# of banks and programs via the -M flag, for example:
#
# csound -+rtmidi=virtual -Mkeyboard.map my_project.csd
#
# INFORMATION ON THE FORMAT
#
# -lines that start with '#' are comments
# -lines that have [] start new bank definitions,
# the contents are bankNum=bankName, with bankNum=[1,16384]
# -lines following bank statements are program definitions
# in the format programNum=programName, with programNum=[1,128]
# -bankNumbers and programNumbers are defined in this file
# starting with 1, but are converted to midi values (starting
# with 0) when read
#
# NOTES
#
# -if an invalid bank definition is found, all program
# definitions that follow will be ignored until a new
# valid bank definition is found
# -if a valid bank is defined by no valid programs found
# for that bank, it will default to General MIDI program
# definitions
# -if an invalid program definition is found, it will be
# ignored

[1=My Bank]
1=My Test Patch 1
2=My Test Patch 2
30=My Test Patch 30

[2=My Bank2]
1=My Test Patch 1(bank2)
2=My Test Patch 2(bank2)
30=My Test Patch 30(bank3)

```

The ten sliders up top are by default set to MIDI Controller number 1-10 though they can be changed to whatever one wishes to use. The controller numbers and values of each slider are set per channel, so one may use different settings and values for each channel.

By default there are 128 banks and for each bank 128 patches defaulting to General Midi names. The MIDI bank standard uses 14-bit resolution to support 16384 possible banks, but the bank numbers by default are 0-127. To use values higher than 127, one should use a custom keyboard map and set the desired bank number value for the bank name. The virtual keyboard will correctly transmit the bank number as MSB and LSB with controller numbers 0 and 32.

Beyond the input available from interacting with the GUI via mouse, one may also trigger off MIDI notes by using the ASCII keyboard when the virtual keyboard window is focused. The layout is done much like a tracker and offers two octaves and a major third to trigger, starting from Middle-C (MIDI note 60). The ASCII keyboard MIDI note values are given in the following table.

**Table 5. ASCII Keyboard MIDI Note Values**

Keyboard Key	MIDI Value
z	60

Keyboard Key	MIDI Value
s	61
x	62
d	63
c	64
v	65
g	66
b	67
h	68
n	69
j	70
m	71
q	72
2	73
w	74
3	75
e	76
r	77
5	78
t	79
6	80
y	81
7	82
u	83
i	84
9	85
o	86
0	87
p	88

Here's an example of usage of the virtual MIDI keyboard. It uses the file *virtual.csd* [examples/virtual.csd].

```

<CsoundSynthesizer>
<CsOptions>; Select audio/midi flags here according to platform
; Audio out  Audio in  Virtual MIDI  -M0 is needed anyway
-odac          -iadc   --rtmidi=virtual -M0
</CsOptions>

<CsInstruments>
; By Mark Jamerson 2007

sr=44100
ksmps=10
nchnls=2

```

```

massign 1,1
prealloc 1,10

instr 1 ;Midi FM synth

inote cpsmidi
iveloc ampmidi 10000
idur = 2
    xtratim 1

kgate oscil 1,10,2
anoise noise 100*inote,.99
acps samphold anoise,kgate
aosc oscili 1000,acps,1
aout = aosc

; Use controller 7 to control volume
kvol ctrl7 1, 7, 0.2, 1

outs kvol * aout, kvol * aout

endin

</CsInstruments>

<CsScore>
f0 3600
f1 0 1024 10 1
f2 0 16 7 1 8 0 8
f3 0 1024 10 1 .5 .6 .3 .2 .5

e
</CsScore>
</CsoundSynthesizer>

```

## MIDI input

The following opcodes can receive MIDI information:

- MIDI information for any instruments: *aftouch*, *chanctrl* and *polyaft*, *pchbend*.
- MIDI information for MIDI-triggered instruments: *veloc*, *midictrl* and *notnum*. See also *Converters*.
- MIDI Controller input for any instrument: *ctrl7*, *ctrl14* and *ctrl21*.
- MIDI Controller input for MIDI-triggered instruments only: *midic7*, *midic14* and *midic21*.
- MIDI controller value initialization: *initc7*, *initc14*, *initc21* and *ctrlinit*.
- Generic MIDI input: *midin*.

*massign* can be used to specify the csound instrument to be triggered by a particular MIDI channel. *pg-massign* can be used to assign a csound instrument to a specific MIDI program.

## MIDI Message Output

Opcodes that produce MIDI output are:

- *mdelay*
- *nrpn*

- *outiat*
- *outic*
- *outic14*
- *outipat*
- *outipb*
- *outipc*
- *outkat*
- *outkc*
- *outkc14*
- *outkpat*
- *outkpb*
- *outkpc*
- *midiout*

## Generic Input and Output

Opcodes for generic MIDI input and output are *midin* and *midout*.

## Converters

The following opcodes can convert MIDI information from a MIDI-triggered instrument instance:

- MIDI note number to frequency converters: *cpsmidi*, *cpsmidib*, *cpstmid*, *octmidi*, *octmidib*, *pchmidi* and *pchmidib*.
- MIDI velocity to amplitude converters: *ampmidi* and *ampmidid*.

## Event Extenders

Opcodes that let one extend the duration of an event are:

- *release*
- *xtratim*

## Note-on/Note-off Output

Opcodes to output MIDI note on or off messages are:

- *midion*
- *midion2*

- *moscil*
- *noteoff*
- *noteon*
- *noteondur*
- *noteondur2*

## MIDI/Score Interoperability opcodes

The following opcodes can be used to design instruments that work interchangeably for real-time MIDI and score events:

- *midichannelaftertouch*
- *midichn*
- *midicontrolchange*
- *mididefault*
- *midinoteoff*
- *midinoteoncps*
- *midinoteonkey*
- *midinoteonoct*
- *midinoteonpch*
- *midipitchbend*
- *midipolyaftertouch*
- *midiprogramchange*.



### Adapting a score-activated Csound instrument.

To adapt an ordinary Csound instrument designed for score activation for score/MIDI interoperability:

- Change all *linen*, *linseg*, and *expseg* opcodes to *linenr*, *linsegr*, and *expsegr*, respectively, except for a de-clicking or damping envelope. This will not materially change score-driven performance.
- Add the following lines at the beginning of the instrument definition:

```
; Ensures that a MIDI-activated instrument
; will have a positive p3 field.
mididefault 60, p3
; Puts MIDI key translated to cycles per
; second into p4, and MIDI velocity into p5
midinoteoncps p4, p5
```

Obviously, *midinoteoncps* could be changed to *midinoteonoct* or any of the other options, and the choice of p-fields is arbitrary.



## MIDI Realtime Input/Output command line options

New *MIDI I/O flags* in Csound 5.02, can replace most uses of these MIDI interop opcodes, and make usage easier.

# System Realtime Messages

Opcodes for System Realtime MIDI messages are: *mclock* and *mrtmsg*.

## Slider Banks

Opcodes for slider banks of MIDI controls are:

- *slider8*
- *slider8f*
- *slider16*
- *slider16f*
- *slider32*
- *slider32f*
- *slider64*
- *slider64f*
- *s16b14*
- *s32b14*
- *sliderKawai*

Opcodes for storing slider banks of MIDI controls to tables are:

- *slider8table*
- *slider8tablef*
- *slider16table*
- *slider16tablef*
- *slider32table*
- *slider32tablef*
- *slider64table*
- *slider64tablef*

---

# Spectral Processing

See the section *Additive Synthesis/Resynthesis* for the basic resynthesis opcodes.

## Short-time Fourier Transform (STFT) Resynthesis



### Use of PVOC-EX files with the old Csound pvoc opcodes

All the original pvoc opcodes can now read a PVOC-EX file, as well as the native non-portable file format. As the PVOC-EX file uses a double-size analysis window, users may find that this gives a useful improvement in quality, for some sounds and processes, despite the fact that the resynthesis does not use the same window size.

Apart from the window size parameter, the main difference between the original .pv format and PVOC-EX is in the amplitude range of analysis frames. While rescaling is applied, so that no significant difference in output level is experienced, whichever file format is used, some slight loss of amplitude can still arise, as the double window usage itself modifies frame amplitudes, of which the resynthesis code is unaware. Note that all the original pvoc opcodes expect a mono analysis file, and multi-channel PVOC-EX files will accordingly be rejected.

Opcodes that implement STFT resynthesis are:

- *mincer*
- *temposcal*
- *tableseg*
- *pvadd*
- *pvbufread*
- *pvcross*
- *pvinterp*
- *pvoc*
- *pvread*
- *tableseg*
- *tablexseg*
- *vpvoc*

Use the utility *PVANAL* to generate pv analysis files.

## Linear Predictive Coding (LPC) Resynthesis

The linear predictive coding resynthesis opcodes are:

- *lpfreson*
- *lpinterp*
- *lpread*
- *lpreson*
- *lpslot*

LPC analysis files can be created using the *LPANAL* utility.

## Non-standard Spectral Processing

These units generate and process non-standard signal data types, such as down-sampled time-domain control signals and audio signals, and their frequency-domain (spectral) representations. The data types (*d-*, *w-*) are self-defining, and the contents are not processable by any other Csound units.

The opcodes for non-standard spectral processing are *specaddm*, *specdiff*, *specdisp*, *specfilt*, *spechist*, *specptrk*, *specscal*, *specsum*, and *spectrum*.

## Tools for Real-time Spectral Processing (pvs opcodes)

With these opcodes, two new core facilities are added to Csound. They offer improved audio quality, and fast performance, enabling high-quality analysis and resynthesis (together with transformations) to be applied in real-time to live signals. The original Csound phase vocoder remains unaltered; the new opcodes use an entirely separate set of functions based on “pvoc.c” in the CARL distribution, written by Mark Dolson.

The Csound *dnoise* and *srconv* utilities (also by Dolson, from CARL) also use this pvoc engine. CARL pvoc is also the basis for the phase vocoder included in the Composer's Desktop Project. A few small but important modifications have been made to the original CARL code to support real-time streaming.

1. Support for the new PVOC-EX analysis file format. This is a fully portable (cross-platform) open file format, supporting three analysis formats, and multi-channel signals. Currently only the standard amplitude+frequency format has been implemented in the opcodes, but the file format itself supports amplitude+phase and complex (real-imaginary) formats. In addition to the new opcodes, the original Csound pvoc opcodes have been extended (and thereby with enhanced audio quality in some cases) to read PVOC-EX files as well as the original (non-portable) format.

Full details of the structure of a PVOC-EX file are available via the website: <http://www.c-s.bath.ac.uk/~jpff/NOS-DREAM/researchdev/pvocex/pvocex.html>. This site also gives details of the freely available console programs *pvocex* and *pvocex2* which can be used to create PVOC-EX files in all supported formats.

2. A new frequency-domain signal type, fully streamable, with *f* as the leading character. In this document it is conveniently referred to as an *fsig*. Primary support for *fsigs* is provided by the opcodes *pvsanal* and *pvsynth*, which perform conventional phase vocoder overlap-add analysis and resynthesis, independently of the orchestra control-rate. The only requirement is that the control-rate *kr* be higher than or equal to the analysis rate, which can be expressed by the requirement that  $ksmps \leq overlap$ , where *overlap* is the distance in samples between analysis frames, as specified for *pvsanal*. As *overlap*

is typically at least 128, and more usually 256, this is not an onerous restriction in practice. The opcode *pvsinfo* can be used at init time to acquire the properties of an *fsig*.

The *fsig* enables the nominal separation between the analysis and resynthesis stages of the phase vocoder to be exposed to the Csound programmer, so that not only can alternatives be employed for either or both of these stages (not only oscillator-bank resynthesis, but also the generation of synthetic *fsig* streams), but opcodes, operating on the *fsig* stream, can themselves become more elemental. Thus the *fsig* enables the creation of a true streaming plugin framework for frequency domain signals. With the old *pvoc* opcodes, each opcode is required to act as a resynthesiser, so that facilities such as pitch scaling are duplicated in each opcode; and in many cases the opcodes are parameter-rich. The separation of analysis and synthesis stages by means of the *fsig* encourages the development of a wide range of simple building-block opcodes implementing one or two functions, with which more elaborate processes can be constructed.

This is very much a preliminary and experimental release, and it is possible that the precise definition of the opcodes may change, in response to user feedback. Also, clearly, many new possibilities for opcodes are opened up; these factors may also have a retrospective influence on the opcodes presented here.

Note that some opcode parameters currently have restricted or missing implementation. This is at least in part in order to keep the opcodes simple at this stage, and also because they highlight important design issues on which no decision has yet been made, and on which opinions from users are sought.

One important point about the new signal type is that because the analysis rate is typically much lower than *kr*, new analysis frames are not available on each *k*-cycle. Internally, the opcodes track *ksmps*, and also maintain a frame counter, so that frames are read and written at the correct times; this process is generally transparent to the user. However, it means that *k*-rate signals only act on an *fsig* at the analysis rate, not at each *k*-cycle. The opcode *pvsftw* returns a *k*-rate flag that is set when new *fsig* data is valid.

Because of the nature of the overlap-add system, the use of these opcodes incurs a small but significant delay, or latency, determined by the window size ( $\max(\text{ifftsize}, \text{iwinsize})$ ). This is typically around 23msecs. In this first release, the delay is slightly in excess of the theoretical minimum, and it is hoped that it can be reduced, as the opcodes are further optimized for real-time streaming.

The opcodes for real-time spectral processing are *pvsadsyn*, *pvsanal*, *pvscross*, *pvsfread*, *pvsftr*, *pvsftw*, *pvsinfo*, *pvsmaska*, and *pvsynth*.

In addition there are a number of opcodes available as plugins in Csound5, and in the core for Csound6. These are *pvstanal*, *pvsdiskin*, *pvscent*, *pvsdemix*, *pvsfreeze*, *pvsbuffer*, *pvsbufread*, *pvsbufread2*, *pvscale*, *pvsift*, *pvsifd*, *pvsinit*, *pvsin*, *pvsout*, *pvsosc*, *pvsbin*, *pvsdisp*, *pvsfwrite*, *pvslock*, *pvmix*, *pvsMOOTH*, *pvsfilter*, *pvsblur*, *pvstencil*, *pvsarp*, *pvsvoc*, *pvsMORPH*, *pvsbandp*, *pvsbandr*, *pvsWARP*, *pvsGAIN*, *pvs2tab*, *tab2pvs*.

A number of opcodes are designed to generate and process streaming partials tracks data. these are *partials*, *trcross*, *trfilter*, *trsplit*, *trmix*, *trscale*, *trshift*, *trlowest*, *trhighest* *tradsyn*, *sinsyn*, *resyn*, *binit*

See the *Stacks* section for information on the stack opcodes which can stack *f*-signals.

## ATS Spectral Processing

These opcodes can read, transform and resynthesize ATS analysis files. Please note that you need the ATS application to produce analysis files. From the ATS Reference Manual:

*"ATS is a software library of functions for spectral Analysis, Transformation, and Synthesis of sound based on a sinusoidal plus critical-band noise model. A sound in ATS is a symbolic object representing a spectral model that can be sculpted using a variety of transformation functions."*

For more information on ATS visit: <http://www-ccrma.stanford.edu/~juan/ATS.html>.

ATS analysis files can be produced using the ATS software or the csound utility *ATSA*.

The opcodes for ATS processing are:

- *ATSinfo*: reads data out of the header of an ATS file.
- *ATSread*, *ATSreadnz*, *ATSbufread*, *ATSinterpread*, *ATSpartialtap*: read data from an ATS file or buffer.
- *ATSadd*, *ATSaddnz*, *ATScross*, *ATSSinnoi*: Resynthesize sound.

## Credits

Author: Alex Norman  
Seattle, Washington  
2004

## Loris Opcodes



### Note

These opcodes are an optional component of Csound5. You can check if they are installed by using the command 'csound -z' which lists all available opcodes.

The Loris family of opcodes wraps: *lorisread* which imports a set of bandwidth-enhanced partials from a SDIF-format data file, applying control-rate frequency, amplitude, and bandwidth scaling envelopes, and stores the modified partials in memory; *lorismorph*, which morphs two stored sets of bandwidth-enhanced partials and stores a new set of partials representing the morphed sound. The morph is performed by linearly interpolating the parameter envelopes (frequency, amplitude, and bandwidth, or noisiness) of the bandwidth-enhanced partials according to control-rate frequency, amplitude, and bandwidth morphing functions, and *lorisplay*, which renders a stored set of bandwidth-enhanced partials using the method of Bandwidth-Enhanced Additive Synthesis implemented in the Loris software, applying control-rate frequency, amplitude, and bandwidth scaling envelopes.

For more information about sound morphing and manipulation using Loris and the Reassigned Bandwidth-Enhanced Additive Model, visit the Loris web site at [www.cerlsoundgroup.org/Loris](http://www.cerlsoundgroup.org/Loris) [<http://www.cerlsoundgroup.org/Loris>].

## Examples

### Example 6. Play the partials without modification

```

;
; Play the partials in clarinet.sdif
; from 0 to 3 sec with 1 ms fadetime
; and no frequency , amplitude, or
; bandwidth modification.
;
instr 1
  ktime      linseg      0, p3, 3.0      ; linear time function from 0 to 3 seconds
             lorisread   ktime, "clarinet.sdif", 1, 1, 1, 1, .001
  asig       lorisplay   1, 1, 1, 1
             out         asig

```

```
endin
```

### Example 7. Add tuning and vibrato

```
; Play the partials in clarinet.sdif
; from 0 to 3 sec with 1 ms fadetime
; adding tuning and vibrato, increasing the
; "breathiness" (noisiness) and overall
; amplitude, and adding a highpass filter.
;
instr 2
  ktime    linseg    0, p3, 3.0    ; linear time function from 0 to 3 seconds

  ; compute frequency scale for tuning
  ; (original pitch was G#4)
  ifscale =          cspch(p4)/cspch(8.08)

  ; make a vibrato envelope
  kenvv    linseg    0, p3/6, 0, p3/6, .02, p3/3, .02, p3/6, 0, p3/6, 0
  kvib     oscil     kenvv, 4, 1    ; table 1, sinusoid

  kbwenv   linseg    1, p3/6, 1, p3/6, 2, 2*p3/3, 2
           lorisread ktime, "clarinet.sdif", 1, 1, 1, 1, .001
  a1       lorisplay 1, ifscale+kvib, 2, kbwenv
  a2       atone     a1, 1000      ; highpass filter, cutoff 1000 Hz
           out       a2
endin
```

The instrument in the first example synthesizes a clarinet tone from beginning to end using partials derived from reassigned bandwidth-enhanced analysis of a three-second clarinet tone, stored in a file, `clarinet.sdif`. The instrument in Example 2 adds tuning and vibrato to the clarinet tone synthesized by instr 1, boosts its amplitude and noisiness, and applies a highpass filter to the result. The following score can be used to test both of the instruments described above.

```
; make sinusoid in table 1
f 1 0 4096 10 1

; play instr 1
;   strt  dur
i 1  0    3
i 1  +    1
i 1  +    6
s

; play instr 2
;   strt  dur  ptch
i 2  1    3    8.08
i 2  3.5  1    8.04
i 2  4    6    8.00
i 2  4    6    8.07
e
```

### Example 8. Morph partials

```
; Morph the partials in clarinet.sdif into the
; partials in flute.sdif over the duration of
; the sustained portion of the two tones (from
; .2 to 2.0 seconds in the clarinet, and from
; .5 to 2.1 seconds in the flute). The onset
; and decay portions in the morphed sound are
; specified by parameters p4 and p5, respectively.
```

```

; The morphing time is the time between the
; onset and the decay. The clarinet partials are
; shifted in pitch to match the pitch of the flute
; tone (D above middle C).
;
instr 1
  ionset =          p4
  idecay =          p5
  itmorph =         p3 - (ionset + idecay)
  ipshift =         cpspch(8.02)/cpspch(8.08)

  ; clarinet time function, morph from .2 to 2.0 seconds
  ktcl  linseg      0, ionset, .2, itmorph, 2.0, idecay, 2.1
  ; flute time function, morph from .5 to 2.1 seconds
  ktfl  linseg      0, ionset, .5, itmorph, 2.1, idecay, 2.3
  kmurph linseg      0, ionset, 0, itmorph, 1, idecay, 1
        lorisread  ktcl, "clarinet.sdif", 1, ipshift, 2, 1, .001
        lorisread  ktfl, "flute.sdif", 2, 1, 1, 1, .001
        lorismorph 1, 2, 3, kmurph, kmurph, kmurph
  asig  lorisplay   3, 1, 1, 1
        out        asig
endin

```

### Example 9. More morphing

```

; Morph the partials in trombone.sdif into the
; partials in meow.sdif. The start and end times
; for the morph are specified by parameters p4
; and p5, respectively. The morph occurs over the
; second of four pitches in each of the sounds,
; from .75 to 1.2 seconds in the flutter-tongued
; trombone tone, and from 1.7 to 2.2 seconds in
; the cat's meow. Different morphing functions are
; used for the frequency and amplitude envelopes,
; so that the partial amplitudes make a faster
; transition from trombone to cat than the frequencies.
; (The bandwidth envelopes use the same morphing
; function as the amplitudes.)
;
instr 2
  ionset =          p4
  imorph =          p5 - p4
  irelease =        p3 - p5

  kttbn  linseg      0, ionset, .75, imorph, 1.2, irelease, 2.4
  ktmeow linseg      0, ionset, 1.7, imorph, 2.2, irelease, 3.4

  kmfreq linseg      0, ionset, 0, .75*imorph, .25, .25*imorph, 1, irelease, 1
  kmamp  linseg      0, ionset, 0, .75*imorph, .9, .25*imorph, 1, irelease, 1

        lorisread  kttbn, "trombone.sdif", 1, 1, 1, 1, .001
        lorisread  ktmeow, "meow.sdif", 2, 1, 1, 1, .001
        lorismorph 1, 2, 3, kmfreq, kmamp, kmamp
  asig  lorisplay   3, 1, 1, 1
        out        asig
endin

```

The instrument in the first morphing example performs a sound morph between a clarinet tone and a flute tone using reassigned bandwidth-enhanced partials stored in `clarinet.sdif` and `flute.sdif`.

The morph is performed over the sustain portions of the tones, 2. seconds to 2.0 seconds in the case of the clarinet tone and .5 seconds to 2.1 seconds in the case of the flute tone. The time index functions, `ktcl` and `ktfl`, align the onset and decay portions of the tones with the specified onset and decay times for the morphed sound, specified by parameters `p4` and `p5`, respectively. The onset in the morphed sounds is

purely clarinet partial data, and the decay is purely flute data. The clarinet partials are shifted in pitch to match the pitch of the flute tone (D above middle C).

The instrument in the second morphing example performs a sound morph between a flutter-tongued trombone tone and a cat's meow using reassigned bandwidth-enhanced partials stored in `trombone.sdif` and `meow.sdif`. The data in these SDIF files have been channelized and distilled to establish correspondences between partials.

The two sets of partials are imported and stored in memory locations labeled 1 and 2, respectively. Both of the original sounds have four notes, and the morph is performed over the second note in each sound (from .75 to 1.2 seconds in the flutter-tongued trombone tone, and from 1.7 to 2.2 seconds in the cat's meow). The different time index functions, `ktbn` and `ktmeow`, align those segments of the source and target partial sets with the specified morph start, morph end, and overall duration parameters. Two different morphing functions are used, so that the partial amplitudes and bandwidth coefficients morph quickly from the trombone values to the cat's-meow values, and the frequencies make a more gradual transition. The morphed partials are stored in a memory location labeled 3 and rendered by the subsequent `lorisplay` instruction. They could also have been used as a source for another morph in a three-way morphing instrument. The following score can be used to test both of the instruments described above.

```

; play instr 1
;   strt dur  onset  decay
i 1  0    3    .25   .15
i 1  +    1    .10   .10
i 1  +    6    1.    1.
s

; play instr 2
;   strt dur  morph_start  morph_end
i 2  0    4    .75         2.75
e

```

## Credits

This implementation of the Loris unit generators was written by Kelly Fitz ([loris@cerlsoundgroup.org](mailto:loris@cerlsoundgroup.org) [<mailto:loris@cerlsoundgroup.org>]).

It is patterned after a prototype implementation of the `lorisplay` unit generator written by Corbin Champion, and based on the method of Bandwidth-Enhanced Additive Synthesis and on the sound morphing algorithms implemented in the Loris library for sound modeling and manipulation. The opcodes were further adapted as a plugin for Csound 5 by Michael Gogins.

## Array-based spectral opcodes



### Note

These opcodes are designed to work with k-rate arrays for spectral data manipulation.

- `fft`,
- `fftin`,
- `rfft`,
- `rfftin`,

- *pvs2array*,
- *pvsfromarray*,
- *cmplxprod*,
- *rect2pol*,
- *pol2rect*,
- *window*,
- *r2c*,
- *c2r*,
- *mags*, and
- *phs*.

---

# Strings

String variables are variables with a name starting with *S* or *gS* (for a local or global string variable, respectively). These variables can be used as input argument to any opcode that expects a quoted string constant, and can be manipulated at initialization or performance time with the opcodes listed below.

It is also possible to use string *p*-fields. The string *p*-field can be used by many orchestra opcodes directly, or it can be copied to a string variable first:

```
a1    diskin2 p5, 1
```

```
Sname strget p5  
a1    diskin2 Sname, 1
```

Strings within *Csound* can be expressed using traditional double quotes (" "), and also using {{ }}. The second method is useful to allow ';' and '\$' characters within the string without having to use ASCII codes.



## Note

String variables and related opcodes are not available in *Csound* versions older than 5.00.

Strings can also be linked to a number using *strset* and *strget*.

*Csound* 5 also has improvements in parsing string constants. It is possible to specify a multi-line string by enclosing it within {{ and }} instead of the usual double quote characters, and the following escape sequences are automatically converted:

- *\a* alert bell
- *\b* backspace
- *\n* new line
- *\r* carriage return
- *\t* tab
- *\\* a single '\' character
- *\nnn* the character of which the ASCII code (in octal) is *nnn*



## Note

If the user does not want an escaped sequence to be automatically converted, she has to escape it with an additional '\' character so that *Csound* knows it does not have to interpret the escaped sequence. For example the string "Not escaped\nline return" will be converted to

```
"Not escaped  
line return"
```

before being used, while the string "Escaped\\nline return" will be converted to

"Escaped\nline return"

before being used.

It can be useful together with the *system* opcode:

```
instr 1
; csound5 lets you make a string with line returns inside double brackets
  system {{      ps
            date
            cd ~/Desktop
            pwd
            ls -l
            whois csounds.com
          }}
endin
```

And the *python* opcodes, among others:

```
pyruni {{
import random

pool = [(1 + i/10.0) ** 1.2 for i in range(100)]

def get_number_from_pool(n, p):
  if random.random() < p:
    i = int(random.random() * len(pool))
    pool[i] = n
  return random.choice(pool)
}}
```

## String Manipulation Opcodes

These opcodes perform operations on string variables (note: most of the opcodes run at init time only, and have a version with a "k" suffix that runs at both init and performance time; exceptions to this rule include puts and strget):

- *strcpy* and *strcpyk* - Assigns to a string variable.
- *strcat* and *strcatk* - Concatenates strings, and stores the result in a variable.
- *strcmp* and *strcmpk* - Compares strings.
- *strget* - Assigns to a string variable, from strset table at the specified index, or string score p-field.
- *strlen* and *strlenk* - Returns the length of a string.
- *sprintf* - printf-style formatted output conversion, storing the result in a string variable.
- *sprintfk* - printf-style formatted output conversion, storing the result in a string variable at k-rate.
- *puts* - Prints a string constant or variable.
- *strindex* and *strindexk* - Returns the first occurrence of a string in another string.
- *strrindex* and *strrindexk* - Returns the last occurrence of a string in another string.
- *strsub* and *strsubk* - Returns a substring of the input string.

## String Conversion Opcodes

These opcodes convert string variables (note: most of the opcodes run at init time only, and have a version with a "k" suffix that runs at both init and performance time; exceptions to this rule include puts and strget):

- *strtod* and *strtodk* - Converts string value to a floating point value at i-rate.
- *strtol* and *strtolk* - Converts string value to signed integer at i-rate.
- *strchar* and *strchark* - Returns the ASCII code of a character in a string.
- *strlower* and *strlowerk* - Converts a string to lower case.
- *strupper* and *strupperk* - Converts a string to upper case.

---

# Vectorial Opcodes

The vectorial opcode family is designed to allow sections of f-tables to be treated as vectors for diverse operations on them.

## Tables of vectors operators

The following Vectorial opcodes support read/write access to arrays of vectors (or arrays of arrays):

- *vtablei*
- *vtablelk*
- *vtablek*
- *vtablea*
- *vtablewi*
- *vtablewk*
- *vtablewa*
- *vtabi*
- *vtabk*
- *vtaba*
- *vtabwi*
- *vtabwk*
- *vtabwa*

## Operations Between a Vectorial and a Scalar Signal

These opcodes perform numeric operations between a vectorial control signal (hosted inside a function table), and a scalar signal. Result is a new vector that overrides old values of the table. There are k-rate and i-rate versions of the opcodes.

All these operators are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2*, etc.

Operations Between a Vectorial and a Scalar Signal:

- *vadd*
- *vmult*
- *vpow*
- *vexp*

- *vadd\_i*
- *vmult\_i*
- *vpow\_i*
- *vexp\_i*

## Operations Between two Vectorial Signals

These opcodes perform operations between two vectors, that is, each element of the first vector is processed with the corresponding element of the other vector. The result is a new vector that overrides the old values of the source vector.

Operations Between two Vectorial Signals:

- *vaddv*
- *vsubv*
- *vmultv*
- *vdivv*
- *vpowv*
- *vexpv*
- *vcopy*
- *vmap*
- *vaddv\_i*
- *vsubv\_i*
- *vmultv\_i*
- *vdivv\_i*
- *vpowv\_i*
- *vexpv\_i*
- *vcopy\_i*

All these operators are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2*, etc.

## Vectorial Envelope Generators

The opcodes to generate vectors containing envelopes are *vlinseg* and *vexpseg*.

These opcodes are similar to *linseg* and *expseg*, but operate with vectorial signals instead of with scalar signals.

Output is a vector hosted by an f-table (that must be previously allocated), while each break-point of the envelope is actually a vector of values. All break-points must contain the same number of elements (*ielements*).

These operators are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2*, etc.

## Limiting and wrapping of vectorial control signals

The opcodes to perform limiting and wrapping of elements within a vector are:

- *vlimit*
- *vwrap*
- *vmirror*

These opcodes are similar to *limit*, *wrap* and *mirror*, but operate on a vector instead of a scalar signal. The old values of the vector contained in an f-table are over-written if they are out of min/max interval. If you want to keep the original values of the input vector, use the *vcopy* opcode to copy it in another table.

All these opcodes work at k-rate.

All these operators are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2* etc.

## Vectorial Control-rate Delay Paths

Vectorial Control-rate Delay Paths:

- *vdelayk*
- *vport*
- *vecdelay*

## Vectorial Random Signal Generators

These opcodes generate vectors of random numbers to be stored in tables. They generate a sort of 'vectorial band-limited noise'. All these opcodes work at k-rate.

Vectorial random signal generators: *vrandh* and *vrandi*.

Cellular automata vectors can be generated using: *vcella*.

---

# Zak Patch System

The zak opcodes are used to create a system for i-rate, k-rate or a-rate patching. The zak system can be thought of as a global array of variables. These opcodes are useful for performing flexible patching or routing from one instrument to another. The system is similar to a patching matrix on a mixing console or to a modulation matrix on a synthesizer. It is also useful whenever an array of variables is required.

The zak system is initialized by the *zakinit* opcode, which is usually placed just after the other global initializations: *sr*, *kr*, *ksmps*, *nchnls*. The *zakinit* opcode defines two areas of memory, one area for i- and k-rate patching, and the other area for a-rate patching. The *zakinit* opcode may only be called once. Once the zak space is initialized, other zak opcodes can be used to read from, and write to the zak memory space, as well as perform various other tasks.

Zak channels count from 0, so if you define 1 channel, the only valid channel is channel 0.

Opcodes for the zak patch system are:

- Audio Rate: *zacl*, *zakinit*, *zamod*, *zar*, *zarg*, *zaw* and *zawm*.
- Control Rate: *zkcl*, *zkmod*, *zkr*, *zkw*, and *zkwm*.
- At initialization: *zir*, *ziw* and *ziwm*

---

# Plugin Hosting

Csound currently hosts external plugins using *dssi4cs* (for LADSPA plugins) on Linux and *vst4cs* (for VST plugins) on Windows and Mac OS X.

## DSSI and LADSPA for Csound

*dssi4cs* enables the use of DSSI and LADSPA plugin effects and synthesizers within Csound on Linux. The following opcodes are available:

- *dssiinit* - Loads a plugin.
- *dssiactivate* - Activates or deactivates a plugin if it has this facility
- *dssilist* - Lists all available plugins found in the LADSPA\_PATH and DSSI\_PATH global variables.
- *dssiaudio* - Process audio using a Plugin.
- *dssictrls* - Send control information to a plugin's control port.

See the entry for *dssiinit* for a usage example.



### Note

Currently only LADSPA plugins are supported, but DSSI support is planned.

## VST for Csound

*vst4cs* enables the use of VST plugin effects and synthesizers within Csound. The following opcodes are available:

- *vstinit* - Loads a plugin.
- *vstaudio*, *vstaudiog* - Returns a plugin's output.
- *vstmidiout* - Sends MIDI data to a plugin.
- *vstparamset*, *vstparamget* - Sends and receives automation data to and from the plugin.
- *vstnote* - Sends a MIDI note with definite duration.
- *vstinfo* - Outputs the Parameter and Program names for a plugin.
- *vstbankload* - Loads an `.fxb` Bank.
- *vstprogset* - Sets a Program in an `.fxb` Bank.
- *vstedit* - Opens the GUI editor for the plugin, when available.

## Credits

By: Andres Cabrera and Michael Gogins

Uses code from Hermann Seib's VSTHost and Thomas Grill's vst~ object.

VST is a trademark of Steinberg Media Technologies GmbH. VST Plug-In Technology by Steinberg.

---

# OSC and Network

## Ableton Link Opcodes

The purpose of Ableton Link is to synchronize musical time, beat, and phase between musical applications performing in real time from separate programs, processes, and network addresses. This is useful, e.g., for laptop orchestras.

Ableton Live is not required to use the Ableton Link protocol, as it is a peer-to-peer protocol. There is one Link session on the local area network that maintains a global time, tempo, and beat. Any peer may set the tempo, and thereafter all peers in the session share that tempo. A process may have any number of peers (i.e., any number of Link objects). Each peer may also define its own "quantum" i.e. some multiple of the beat, e.g. a quantum of 4 might imply 1 beat every measure of 4/4 time. The phase of the time is defined w.r.t the quantum, e.g. phase 0.5 of a quantum of 4 would be the second beat of the measure. Peers may read and write timelines with local time, beat, and phase, counting from when the peer is enabled, but the tempo and beat on all timelines for all peers in the session will coincide.

The first peer in a session determines the initial tempo. After that, the tempo is changed only, and whenever, any peer explicitly calls the set tempo function (`link_tempo_set`, in Csound).

The Link tempo is independent of the Csound score tempo. Performances that need to synchronize the score tempo with the Link tempo may use the tempo opcode to set the score tempo from the Link tempo; or conversely, set the Link tempo from the score tempo using the `tempoval` opcode.

Please note, the phase and beat obtained or set by these opcodes is only as precise as allowed by the duration of Csound's `kperiod`, the audio driver used by Csound, network latency and stability, and the system's most precise clock.

- `link_create` - Creates an Ableton Link peer object.
- `link_enable` - Enable/disable synchronization with the network Ableton Link session tempo and beat.
- `link_is_enabled` - Returns whether or not this Ableton Link peer has joined the network session.
- `link_tempo_set` - Sets the tempo for the network's Ableton Link session.
- `link_tempo_get` - Returns the tempo of the network's Ableton Link session.
- `link_beat_get` - Returns the beat, phase, and current time of Ableton Link for this session for a given quantum.
- `link_metro` - Returns a trigger that is 1 on the beat and 0 otherwise along with the beat, phase, and current time of Ableton Link for this session for a given quantum.
- `link_beat_request` - Requests a beat with a specific number at a specific time at a given quantum.
- `link_beat_force` - Forces the network Ableton Link session to adopt a beat with a specific number at a specific time at a given quantum.
- `link_peers` - Returns the number of peers currently joined in the network Ableton Link session.

## OSC

OSC enables interaction between different audio processes, and in particular between Csound and other synthesis engines. The following opcodes are available:

- *OSCinit* - Start an OSC listener thread.
- *OSClisten* - Receive OSC messages.
- *OSCsend* - Send an OSC message.

## Credits

By: John ffitch with the liblo library as inspiration and support.

## Network

The following opcodes can stream or receive audio through UDP:

- *sockrecv*
- *socksend*

## Remote Opcodes

The Remote opcodes enable transmission of score or MIDI events through a network, so remote instances (or a different local instance) can process them. The following opcodes are available:

- *insglobal* - Used to implement a remote orchestra.
- *insremot* - Used to implement a remote orchestra.
- *midiglobal* - Used to implement a remote MIDI orchestra.
- *midiremot* - Used to implement a remote MIDI orchestra.
- *remoteport* - Defines the port for use with the remote system.

---

# Mixer Opcodes

The Mixer family of opcodes provides a global mixer for Csound. The Mixer opcodes include *MixerSend* for sending (that is, mixing in) an arate signal from any instrument to a channel of a mixer buss, *MixerReceive* for receiving an arate signal from a channel of any mixer buss in any instrument, *MixerSetLevel* (krate) and *MixerSetLevel\_i* (irate) for controlling the level of the signal sent from a particular send to a particular buss, *MixerGetLevel* for reading (at krate) the level for sending a signal from a particular send to a particular buss, and *MixerClear* for resetting the busses to zero before the next kperiod of a performance.

---

# Signal Flow Graph Opcodes

These opcodes enable the use of signal flow graphs (AKA asynchronous data flow graphs) in Csound orchestras. Signals flow from the outlets of source instruments and are summed in the inlets of sink instruments. Signals may be krate, arate, frate, or arate arrays. Any number of outlets may be connected to any number of inlets. When a new instance of an instrument is instantiated during performance, the declared connections also are automatically instantiated.

Signal flow graphs simplify the construction of complex mixers, signal processing chains, and the like. They also simplify the re-use of "plug and play" instrument definitions and even entire sub-orchestras, which can simply be `#included` and then "plugged in" to existing orchestras.

Note that inlets and outlets are defined in instruments without reference to how they are connected. Connections are defined in the orchestra header. It is this separation that enables plug-in instruments.

Inlets must be named. Instruments may be named or numbered, but in either case each source instrument must be defined in the orchestra before any of its sinks. Naming instruments makes it easier to connect outlets and inlets in any higher-level orchestra to inlets and outlets in any lower-level `#included` orchestra.

The signal flow graph opcodes include: *outleta*, for sending an arate signal from any instrument out a named port. *outletk*, for sending a krate signal from any instrument out a named port. *outletkid*, similar to *outletk*, but receiving a krate signal only from an identified instance of a port. *outletf*, for sending an frate signal from any instrument out a named port. *outletv*, for sending an arate array signal from any instrument out a named port. *inleta*, for receiving an arate signal through a named port. *inletk*, for receiving a krate signal through a named port. *inletkid*, similar to *inletk*, but transmitting a signal only between inlet and outlet opcodes. *inletf*, for receiving an frate signal through a named port. *inletv*, for receiving an arate array signal through a named port. *connect*, for routing the signal from a named outlet in a source instrument to a named inlet in a sink instrument. *alwayson* for permanently activating an instrument from the orchestra header, without need of a score statement, e.g. for use as an effect processor receiving inputs from a number of sources. *ftgenonce* for instantiating function tables from within instrument definitions, without need for *f*-statements in the score or *ftgen* opcodes in the orchestra header.

A typical scenario for the use of these opcodes would be something like this. A set of instruments would be defined, each in its own orchestra file, and each instrument would define inlet ports, outlet ports, and function tables within itself. Such instruments are completely self-contained. Then, a set of effects processors, such as equalizers, reverbs, compressors, and so on, would also be defined, each in its own file. Then, a customized master orchestra would `#include` the instruments and effects to be used, route the outputs of some instruments into one equalizer and the outputs of other effects into another equalizer, then route the outputs of both equalizers into a reverb, the output of the reverb into a compressor, and the output of the compressor into a stereo output soundfile.

## Example

Here is an example of the signal flow graph opcodes. It uses the file *signalflowgraph.csd* [examples/signalflowgraph.csd].

### Example 10. Example of the signal flow graph opcodes.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o madsr.wav -W ;; for file output any platform
```

```

</CsOptions>
<CsInstruments>

/* Written by Michael Gogins */
; Initialize the global variables.
sr = 44100
ksmps = 100
nchnls = 2

; Connect up the instruments to create a signal flow graph.

connect "SimpleSine", "leftout", "Reverberator", "leftin"
connect "SimpleSine", "rightout", "Reverberator", "rightin"

connect "Moogy", "leftout", "Reverberator", "leftin"
connect "Moogy", "rightout", "Reverberator", "rightin"

connect "Reverberator", "leftout", "Compressor", "leftin"
connect "Reverberator", "rightout", "Compressor", "rightin"

connect "Compressor", "leftout", "Soundfile", "leftin"
connect "Compressor", "rightout", "Soundfile", "rightin"

; Turn on the "effect" units in the signal flow graph.

alwayson "Reverberator", 0.91, 12000
alwayson "Compressor"
alwayson "Soundfile"

instr SimpleSine
  ihz = cpsmidinn(p4)
  iamplitude = ampdb(p5)
  print ihz, iamplitude
  ; Use ftgenonce instead of ftgen, ftgentmp, or f statement.
  isine ftgenonce 0, 0, 4096, 10, 1
  a1 oscili iamplitude, ihz, isine
  aenv madsr 0.05, 0.1, 0.5, 0.2
  asignal = a1 * aenv
  ; Stereo audio outlet to be routed in the orchestra header.
  outleta "leftout", asignal * 0.25
  outleta "rightout", asignal * 0.75
endin

instr Moogy
  ihz = cpsmidinn(p4)
  iamplitude = ampdb(p5)
  ; Use ftgenonce instead of ftgen, ftgentmp, or f statement.
  isine ftgenonce 0, 0, 4096, 10, 1
  asignal vco iamplitude, ihz, 1, 0.5, isine
  kfco line 200, p3, 2000
  krez init 0.9
  asignal moogvcf asignal, kfco, krez, 100000
  ; Stereo audio outlet to be routed in the orchestra header.
  outleta "leftout", asignal * 0.75
  outleta "rightout", asignal * 0.25
endin

instr Reverberator
  ; Stereo input.
  aleftin inleta "leftin"
  arightin inleta "rightin"
  idelay = p4
  icutoff = p5
  aleftout, arightout reverbsc aleftin, arightin, idelay, icutoff
  ; Stereo output.
  outleta "leftout", aleftout
  outleta "rightout", arightout

```

```
endin

instr Compressor
; Stereo input.
aleftin inleta "leftin"
arightin inleta "rightin"
kthreshold = 25000
icomp1 = 0.5
icomp2 = 0.763
irtime = 0.1
iftime = 0.1
aleftout dam aleftin, kthreshold, icomp1, icomp2, irtime, iftime
arightout dam arightin, kthreshold, icomp1, icomp2, irtime, iftime
; Stereo output.
outleta "leftout", aleftout
outleta "rightout", arightout
endin

instr Soundfile
; Stereo input.
aleftin inleta "leftin"
arightin inleta "rightin"
outs aleftin, arightin
endin

</CsInstruments>
<CsScore>
; Not necessary to activate "effects" or create f-tables in the score!
; Overlapping notes to create new instances of instruments.
i "SimpleSine" 1 5 60 85
i "SimpleSine" 2 5 64 80
i "Moogy" 3 5 67 75
i "Moogy" 4 5 71 70
e 1
</CsScore>
</CsoundSynthesizer>
```

---

# Jacko Opcodes

These opcodes enable the use of Jack ports from within Csound orchestras and instruments. Ports can receive or send audio or MIDI data, and send note data.

The Jacko opcodes do not replace the Jack driver and Jack command-line options for Csound, nor do the Jacko opcodes work with them (hence the name "Jacko" instead of "Jack"). The Jacko opcodes are an independent facility that offers greater flexibility in signal routing.

In addition, the Jacko opcodes can work with the Jack system in "freewheeling" mode, which enables the use of Jack-enabled external synthesizers, such as Aeolus or Pianoteq, to render Csound pieces either faster or, even more importantly, slower than real time. This is extremely useful for rendering complex pieces without dropouts using instruments, such as Aeolus, that may not be available except through Jack.

The Jacko opcodes include: *JackoInit*, for initializing the current instance of Csound as a Jack client. *JackoInfo*, for printing information about the Jack daemon, its clients, their ports, and their connections. *JackoFreewheel*, for turning Jack's freewheeling mode on or off. *JackoAudioInConnect*, for creating a connection from an external Jack audio output port to a Jack port in Csound. *JackoAudioOutConnect*, for creating a connection from a Jack port in Csound to an external Jack audio input port. *JackoMidiInConnect*, for creating a connection from an external Jack MIDI port. MIDI events from Jack are received by Csound's regular MIDI opcodes and MIDI interop system. *JackoMidiOutConnect*, for creating a connection from a Jack port in Csound to an external Jack MIDI input port. *JackoOn*, for turning Jack ports in Csound on or off. *JackoAudioIn*, for receiving audio from a Jack input port in Csound, which in turn has received the audio from its connected external port. *JackoAudioOut*, for sending audio to a Jack output port in Csound, which in turn will send the audio on to its connected external port. *JackoMidiOut*, for sending MIDI channel messages to a Jack output port in Csound, which in turn will send the MIDI on to its connected external port. *JackoNoteOut*, for sending a note (with duration) to a Jack output port in Csound, which in turn will send the note on to its connected external port. *JackoTransport*, for controlling the Jack transport.

A typical scenario for the use of the Jacko opcodes would be something like this.

## Example

Here is an example of the Jacko opcodes. It uses the file *jacko.csd* [examples/jacko.csd].

### Example 11. Example of the Jacko opcodes.

```
<CsoundSynthesizer>
<CsOptions>
csound -m255 -M0 -+rtmidi=null -Rwf --midi-key=4 --midi-velocity=5 -o jacko_test.wav
</CsOptions>
<CsInstruments>

;;;
;;; NOTE: this csd must be run after starting "aeolus -t".
;;;

sr          = 48000
; The control rate must be BOTH a power of 2 (for Jack)
; AND go evenly into sr. This is about the only one that works!
ksmps      = 128
nchnls     = 2
0dbfs      = 1

JackoInit  "default", "csound"

; To use ALSA midi ports, use "jackd -Xseq"
```

## Jacko Opcodes

---

```
; and use "jack_lsp -A -c" or aliases from JackInfo,
; probably together with information from the sequencer,
; to figure out the damn port names.

; JackoMidiInConnect  "alsa_pcm:in-131-0-Master", "midiin"
JackoAudioInConnect  "aeolus:out.L", "leftin"
JackoAudioInConnect  "aeolus:out.R", "rightin"
JackoMidiOutConnect  "midiout", "aeolus:Midi/in"

; Note that Jack enables audio to be output to a regular
; Csound soundfile and, at the same time, to a sound
; card in real time to the system client via Jack.

JackoAudioOutConnect "leftout", "system:playback_1"
JackoAudioOutConnect "rightout", "system:playback_2"
JackoInfo

; Turning freewheeling on seems automatically
; to turn system playback off. This is good!

JackoFreewheel 1
JackoOn

alwayson "jackin"

instr 1
; //////////////////////////////////////
ichannel = p1 - 1
itime = p2
iduration = p3
ikey = p4
ivelocity = p5
JackoNoteOut "midiout", ichannel, ikey, ivelocity
print itime, iduration, ichannel, ikey, ivelocity
endin

instr jackin
; //////////////////////////////////////
JackoTransport 3, 1.0
aleft JackoAudioIn "leftin"
aright JackoAudioIn "rightin"

; Aeolus uses MIDI controller 98 to control stops.
; Only 1 data value byte is used, not the 2 data
; bytes often used with NRPNS.
; The format for control mode is 01mm0ggg:
; mm 10 to set stops, 0, ggg group (or Division, 0 based).
; The format for stop selection is 000bbbb:
; bbbbb for button number (0 based).

; Mode to enable stops for Divison I: b1100010 (98
; [this controller VALUE is a pure coincidence]).

JackoMidiOut "midiout", 176, 0, 98, 98

; Stops: Principal 8 (0), Principal 4 (1) , Flote 8 (8) , Flote 2 (10)

JackoMidiOut "midiout", 176, 0, 98, 0
JackoMidiOut "midiout", 176, 0, 98, 1
JackoMidiOut "midiout", 176, 0, 98, 8
JackoMidiOut "midiout", 176, 0, 98, 10

; Sends audio coming in from Aeolus out
; not only to the Jack system out (sound card),
; but also to the output soundfile.
; Note that in freewheeling mode, "leftout"
; and "rightout" simply go silent.
```

```
    JackoAudioOut "leftout", aleft
    JackoAudioOut "rightout", aright
    outs    aright, aleft
    endin

</CsInstruments>
<CsScore>
f 0 30
i 1 1 30 60 60
i 1 2 30 64 60
i 1 3 30 71 60
e 2
</CsScore>
</CsoundSynthesizer>
```

## Credits

By: Michael Gogins 2010

---

# Lua Opcodes

The purposes of the Lua opcodes are:

1. Make it possible to write Csound code in a user-friendly, high-level language with full lexical scoping, structures and classes, and support for functional programming, using LuaJIT (the Lua programming language, implemented with a just-in-time compiler and foreign function interface).
2. Require the installation of no third party software packages, or at least a minimum installation; also, require no build system or external compilation.
3. Run *really fast*; typically, almost as fast as compiled C code, and several times faster than user-defined opcodes.

Using the Lua opcode family, you can interact with the Lua interpreter and just-in-time compiler (luajit) embedded in Csound as follows:

1. Execute any arbitrary block of Lua code (the `lua_exec` opcode),
2. Define an opcode in Lua taking any number or type of parameters, and returning any number or type of parameters (the `lua_opdef` opcode),
3. Call a Lua opcode at i-rate (the `lua_iopcall` opcode),
4. Call a Lua opcode at i-rate and k-rate (the `lua_ikopcall` opcodes), or
5. Call a Lua opcode at i-rate and a-rate (the `lua_iaopcall` opcode).

Lua is Portuguese for "moon." And Lua (<http://www.lua.org>) is a lightweight, efficient dynamic programming language, designed for embedding in C/C++ and extending with C/C++. Lua has a stack-based calling mechanism and provides a toolkit of features (tables, metatables, anonymous functions, and closures) with which many styles of object-oriented and functional programming may be implemented. Lua's syntax is only slightly harder than Python's.

Lua is already one of the fastest dynamic languages; yet LuaJIT by Mike Pall (<http://luajit.org>) goes much further, giving Lua a just-in-time optimizing trace compiler for Intel architectures. LuaJIT includes an efficient foreign function interface (FFI) with the ability to define C arrays, structures, and other types in Lua. The speed of LuaJIT/FFI ranges from several times as fast as Lua, to faster (in some contexts) than optimized C.

## Example

Here is an example of a Lua opcode, implementing a Moog ladder filter. For purposes of comparison, a user-defined opcode and the native Csound opcode that compute the same sound using the same algorithm also are shown, and timed.. The example uses the file `luamoog.csd` [examples/luamoog.csd].

### Example 12. Example of a Lua opcode.

```
<CsoundSynthesizer>
<CsInstruments>
sr = 48000
ksmps = 100
nchnls = 1

gibegan    rtclock

lua_opdef  "moogladder", {{
```

```

local ffi = require("ffi")
local math = require("math")
local string = require("string")
local csoundApi = ffi.load('csound64.dll.5.2')
ffi.cdef[[
    int csoundGetKsmps(void *);
    double csoundGetSr(void *);
    struct moogladder_t {
        double *out;
        double *inp;
        double *freq;
        double *res;
        double *istor;
        double sr;
        double ksmps;
        double thermal;
        double f;
        double fc;
        double fc2;
        double fc3;
        double fcr;
        double acr;
        double tune;
        double res4;
        double input;
        double i;
        double j;
        double k;
        double kk;
        double stg[6];
        double delay[6];
        double tanhstg[6];
    };
]]

local moogladder_ct = ffi.typeof('struct moogladder_t *')

function moogladder_init(csound, opcode, arguments)
    local p = ffi.cast(moogladder_ct, arguments)
    p.sr = csoundApi.csoundGetSr(csound)
    p.ksmps = csoundApi.csoundGetKsmps(csound)
    if p.istor[0] == 0 then
        for i = 0, 5 do
            p.delay[i] = 0.0
        end
        for i = 0, 3 do
            p.tanhstg[i] = 0.0
        end
    end
    return 0
end

function moogladder_kontrol(csound, opcode, arguments)
    local p = ffi.cast(moogladder_ct, arguments)
    -- transistor thermal voltage
    p.thermal = 1.0 / 40000.0
    if p.res[0] < 0.0 then
        p.res[0] = 0.0
    end
    -- sr is half the actual filter sampling rate
    p.fc = p.freq[0] / p.sr
    p.f = p.fc / 2.0
    p.fc2 = p.fc * p.fc
    p.fc3 = p.fc2 * p.fc
    -- frequency & amplitude correction
    p.fcr = 1.873 * p.fc3 + 0.4955 * p.fc2 - 0.6490 * p.fc + 0.9988
    p.acr = -3.9364 * p.fc2 + 1.8409 * p.fc + 0.9968
end

```

```

-- filter tuning
p.tune = (1.0 - math.exp(-(2.0 * math.pi * p.f * p.fcr))) / p.thermal
p.res4 = 4.0 * p.res[0] * p.acr
-- Nested 'for' loops crash, not sure why.
-- Local loop variables also are problematic.
-- Lower-level loop constructs don't crash.
p.i = 0
while p.i < p.ksmps do
  p.j = 0
  while p.j < 2 do
    p.k = 0
    while p.k < 4 do
      if p.k == 0 then
        p.input = p.inp[p.i] - p.res4 * p.delay[5]
        p.stg[p.k] = p.delay[p.k] + p.tune * (math.tanh(p.input * p.thermal) - p.tanhstg[p.k])
      else
        p.input = p.stg[p.k - 1]
        p.tanhstg[p.k - 1] = math.tanh(p.input * p.thermal)
        if p.k < 3 then
          p.kk = p.tanhstg[p.k]
        else
          p.kk = math.tanh(p.delay[p.k] * p.thermal)
        end
        p.stg[p.k] = p.delay[p.k] + p.tune * (p.tanhstg[p.k - 1] - p.kk)
      end
      p.delay[p.k] = p.stg[p.k]
      p.k = p.k + 1
    end
    -- 1/2-sample delay for phase compensation
    p.delay[5] = (p.stg[3] + p.delay[4]) * 0.5
    p.delay[4] = p.stg[3]
    p.j = p.j + 1
  end
  p.out[p.i] = p.delay[5]
  p.i = p.i + 1
end
return 0
end
}}

/*
Moogladder - An improved implementation of the Moog ladder filter

DESCRIPTION
This is a new digital implementation of the Moog ladder filter based on the work of Antti Huovilainen,
described in the paper "Non-Linear Digital Implementation of the Moog Ladder Filter" (Proceedings of
This implementation is probably a more accurate digital representation of the original analogue filter.
This is version 2 (revised 14/DEC/04), with improved amplitude/resonance scaling and frequency correcti

SYNTAX
ar Moogladder asig, kcf, kres

PERFORMANCE
asig - input signal
kcf - cutoff frequency (Hz)
kres - resonance (0 - 1).

CREDITS
Victor Lazzarini
*/

opcode moogladderu, a, akk
asig, kcf, kres      xin
                    setksmps 1
ipi                  =      4 * taninv(1)
/* filter delays */
azl                  init    0

```

```

az2          init          0
az3          init          0
az4          init          0
az5          init          0
ay4          init          0
amf          init          0
              if           kres > 1 then
kres         =             1
              elseif      kres < 0 then
kres         =             0
              endif
/* twice the \'thermal voltage of a transistor\' */
i2v          =             40000
/* sr is half the actual filter sampling rate */
kfc          =             kcf/sr
kf           =             kcf/(sr*2)
/* frequency & amplitude correction */
kfcrcr      =             1.8730 * (kfc^3) + 0.4955 * (kfc^2) - 0.6490 * kfc + 0.9988
kacrcr      =             -3.9364 * (kfc^2) + 1.8409 * kfc + 0.9968;
/* filter tuning */
k2vg        =             i2v * (1 - exp(-2 * ipi * kfcrcr * kf))
/* cascade of 4 1st order sections */
ay1         =             az1 + k2vg * (tanh((asig - 4 * kres * amf * kacrcr) / i2v) - tanh(az1 / i2v))
ay1         =             ay1
ay2         =             az2 + k2vg * (tanh(ay1 / i2v) - tanh(az2 / i2v))
ay2         =             ay2
ay3         =             az3 + k2vg * (tanh(ay2 / i2v) - tanh(az3 / i2v))
ay3         =             ay3
ay4         =             az4 + k2vg * (tanh(ay3 / i2v) - tanh(az4 / i2v))
ay4         =             ay4
/* 1/2-sample delay for phase compensation */
amf         =             (ay4 + az5) * 0.5
ay5         =             ay4
/* oversampling */
ay1         =             az1 + k2vg * (tanh((asig - 4 * kres * amf * kacrcr) / i2v) - tanh(az1 / i2v))
ay1         =             ay1
ay2         =             az2 + k2vg * (tanh(ay1 / i2v) - tanh(az2 / i2v))
ay2         =             ay2
ay3         =             az3 + k2vg * (tanh(ay2 / i2v) - tanh(az3 / i2v))
ay3         =             ay3
ay4         =             az4 + k2vg * (tanh(ay3 / i2v) - tanh(az4 / i2v))
ay4         =             ay4
amf         =             (ay4 + az5) * 0.5
ay5         =             ay4
xout        =             amf
endop

instr 1
              prints      "No filter.\n"
kfe         expseg       500, p3*0.9, 1800, p3*0.1, 3000
              kenv        linen      10000, 0.05, p3, 0.05
              asig        buzz       kenv, 100, sr/(200), 1
              ; afile     moogladder asig, kfe, 1
              out         asig

endin

instr 2
              prints      "Native moogladder.\n"
kfe         expseg       500, p3*0.9, 1800, p3*0.1, 3000
              kenv        linen      10000, 0.05, p3, 0.05
              asig        buzz       kenv, 100, sr/(200), 1
              afile       moogladder asig, kfe, 1
              out         afile

endin

instr 3
              prints      "UDO moogladder.\n"

```

```

kfe      expseg      500, p3*0.9, 1800, p3*0.1, 3000
kenv     linen      10000, 0.05, p3, 0.05
asig     buzz       kenv, 100, sr/(200), 1
afil     moogladderu asig, kfe, 1
          out        afil

endin

instr 4
          prints     "Lua moogladder.\n"
kres     init       1
istor    init       0
kfe      expseg      500, p3*0.9, 1800, p3*0.1, 3000
kenv     linen      10000, 0.05, p3, 0.05
asig     buzz       kenv, 100, sr/(200), 1
afil     init       0
          lua_ikopcall "moogladder", afil, asig, kfe, kres, istor
          out        afil

endin

instr 5
          giended    rtclock
          ielapsed   =          giended - gibegan
          gibegan    print     ielapsed
          gibegan    rtclock

endin

</CsInstruments>

<CsScore>
f 1      0 65536 10 1
i 5.1    0 1
i 4      1 20
i 5.2    21 1
i 4      22 20
i 5.3    42 1
i 2      43 20
i 5.4    63 1
i 2      64 20
i 5.5    84 1
i 3      85 20
i 5.6    105 1
i 3      106 20
i 5.7    126 1
i 1      127 20
i 5.8    147 1
i 1      148 20
i 5.9    168 1
i 4      169 20
i 4      170 20
i 4      171 20
e
</CsScore>

</CsoundSynthesizer>

```

## Credits

Copyright (c) 2011 by Michael Gogins. All rights reserved.

---

# Python Opcodes

## Introduction

Using the Python opcode family, you can interact with a Python interpreter embedded in Csound in five ways:

1. Initialize the Python interpreter (the *pyinit* opcodes),
2. Run a statement (the *pyrun* opcodes),
3. Execute a script (the *pyexec* opcodes),
4. Invoke a callable and pass arguments (the *pycall* opcodes),
5. Evaluate an expression (the *pyeval* opcodes), or
6. Change the value of a Python object, possibly creating a new Python object (the *pyassign* opcodes);

and you can do any of these things:

1. At i-time or at k-time,
2. In the global Python namespace, or in a namespace specific to an individual instance of a Csound instrument (local or "l" context),
3. And can you can retrieve from 0 to 8 return values from callables that accept N parameters.

...this means that there are many Python-related opcodes. But all of these opcodes share the same *py* prefix, and have a regular naming scheme:

```
"py" + [optional context prefix] + [action name] + [optional x-time suffix]
```

## Orchestra Syntax

Blocks of Python code, and indeed entire scripts, can be embedded in Csound orchestras using the `{{` and `}}` directives to enclose the script, as follows:

```
sr=44100
kr=4410
ksmps=10
nchnls=1
pyinit

giSinusoid ftgen 0, 0, 8192, 10, 1

pyruni {{
import random

pool = [(1 + i/10.0) ** 1.2 for i in range(100)]

def get_number_from_pool(n, p):
    if random.random() < p:
        i = int(random.random() * len(pool))
        pool[i] = n
    return random.choice(pool)
```

```
}}  
  
instr 1  
  k1 oscil 1, 3, giSinusoid  
  k2 pycall1 "get_number_from_pool", k1 + 2, p4  
  printk 0.01, k2  
endin
```

## Credits

Copyright (c) 2002 by Maurizio Umberto Puxeddu. All rights reserved.

Portions copyright (c) 2004 and 2005 by Michael Gogins.

---

# Image processing opcodes

Here is a list of opcodes that read/write image files:

- *imagecreate*
- *imagesize*
- *imagegetpixel*
- *imagesetpixel*
- *imagesave*
- *imageload*
- *imagefree*

---

# STK Opcodes



## Note

These opcodes are an optional component of Csound6. You can check if they are installed by using the command 'csound -z' which lists all available opcodes.

The STK opcodes can be built in several ways. If you are already building Csound, simply follow the directions at the top of the Opcodes/stk/stkOpcodes.cpp file.

The STK family of opcodes wraps:

- *STKBandedWG*
- *STKBeeThree*
- *STKBlowBotl*
- *STKBlowHole*
- *STKBowed*
- *STKBrass*
- *STKClarinet*
- *STKDrummer*
- *STKFlute*
- *STKFMVoices*
- *STKHevyMetl*
- *STKMandolin*
- *STKModalBar*
- *STKMoog*
- *STKPercFlut*
- *STKPlucked*
- *STKResonate*
- *STKRhodey*
- *STKSaxofony*
- *STKShakers*
- *STKSimple*
- *STKSitar*
- *STKStifKarp*

- *STKTubeBell*
- *STKVoicForm*
- *STKWhistle*
- *STKWurley*

For more information about the STK opcodes, visit The Synthesis ToolKit in C++ (STK) web site at <https://ccrma.stanford.edu/software/stk>.

## Credits

This implementation of the STK unit generators was written by Perry R. Cook and Gary P. Scavone.

The opcodes were further adapted as a plugin for Csound 5 by Michael Gogins.

---

# Miscellaneous opcodes

Here is a list of opcodes that don't fall in any category:

- *system* - Call an external program via the system call.
- *modmatrix* - modulation matrix opcode with optimizations for sparse matrices.

---

## **Part III. Reference**

---

---

## Table of Contents

Orchestra Opcodes and Operators .....	210
!= .....	211
#define .....	213
#include .....	217
#undef .....	219
#ifdef .....	220
#ifndef .....	221
\$NAME .....	222
% .....	225
&& .....	227
> .....	229
>= .....	231
< .....	233
<= .....	235
* .....	237
+ .....	240
- .....	243
/ .....	246
= .....	249
+= .....	251
== .....	253
^ .....	255
.....	257
Odbfs .....	260
A4 .....	263
<< .....	264
>> .....	266
& .....	267
.....	269
¬ .....	270
# .....	271
a .....	272
abs .....	274
active .....	276
adsr .....	280
adsyn .....	283
adsynt .....	285
adsynt2 .....	288
aftouch .....	291
alpass .....	293
alwayson .....	295
ampdb .....	298
ampdbfs .....	300
ampmidi .....	302
ampmidid .....	304
areson .....	306
aresonk .....	308
atone .....	310
atonek .....	312
atonex .....	314
ATSadd .....	316

---

ATSaddnz .....	320
ATSbufread .....	323
ATScross .....	325
ATSinfo .....	328
ATSinterpread .....	331
ATSread .....	333
ATSreadnz .....	337
ATSpartialtap .....	340
ATSinnoi .....	342
babo .....	346
balance .....	350
bamboo .....	352
barmodel .....	354
bbcutm .....	356
bbcuts .....	361
betarand .....	364
bexprnd .....	367
bformenc1 .....	369
bformdec1 .....	372
binit .....	375
biquad .....	377
biquada .....	382
birnd .....	385
bpf .....	387
bqrez .....	389
butbp .....	391
butbr .....	392
buthp .....	393
butlp .....	394
butterbp .....	395
butterbr .....	397
butterhp .....	399
butterlp .....	401
button .....	403
buzz .....	405
c2r .....	407
cabasa .....	409
cauchy .....	411
cauchy1 .....	413
ceil .....	415
cell .....	417
cent .....	420
centroid .....	422
ceps .....	424
cepsinv .....	426
cggoto .....	428
chanctrl .....	430
changed .....	432
changed2 .....	434
chani .....	437
chano .....	438
cbrt .....	439
chebyshevpoly .....	441
checkbox .....	444

---

chn	446
chnclear	448
chnexport	450
chnget	452
chnmix	455
chnparams	457
chnset	458
chuap	461
cigoto	465
ckgoto	467
clear	469
clfilt	471
clip	474
clockoff	476
clockon	478
cmp	480
cmplxprod	482
cngoto	484
comb	486
combinv	488
compilecsd	490
compileorc	492
compilestr	494
compress	496
compress2	499
connect	502
control	505
convle	506
convolve	507
copya2ftab	511
copyf2array	513
cos	515
cosseg	517
cossegb	519
cossegr	521
cosh	523
cosinv	525
cps2pch	527
cpsmidi	531
cpsmidib	533
cpsmidinn	535
cpsoct	539
cpspch	542
cpstmid	545
cpstun	547
cpstuni	550
cpsxpch	553
cpumeter	557
cpuprc	559
cross2	562
crossfm	564
crunch	567
ctrl14	569
ctrl21	571

---

ctrl7 .....	573
ctrlinit .....	576
cuserrnd .....	577
dam .....	580
date .....	583
dates .....	585
db .....	587
dbamp .....	589
dbfsamp .....	591
dcblock .....	593
dcblock2 .....	595
dconv .....	597
dct .....	599
dctinv .....	601
delay .....	603
delay1 .....	605
delayk .....	607
delayr .....	610
delayw .....	612
deltap .....	614
deltap3 .....	617
deltapi .....	620
deltapn .....	623
deltapx .....	625
deltapxw .....	627
denorm .....	629
diff .....	631
diode_ladder .....	633
directory .....	636
diskgrain .....	638
diskin .....	641
diskin2 .....	644
dispfft .....	648
display .....	650
distort .....	652
distort1 .....	654
divz .....	656
doppler .....	658
dot .....	660
downsamp .....	661
dripwater .....	663
dssiactivate .....	665
dssiaudio .....	668
dssictls .....	670
dssiinit .....	672
dssilist .....	674
dumpk .....	676
dumpk2 .....	679
dumpk3 .....	682
dumpk4 .....	685
duserrnd .....	688
dust .....	690
dust2 .....	692
else .....	694

---

elseif .....	696
endif .....	698
endin .....	700
endop .....	702
envlpx .....	705
envlpxr .....	708
ephasor .....	711
eqfil .....	713
evalstr .....	715
event .....	716
event_i .....	720
exciter .....	722
exitnow .....	724
exp .....	726
expcurve .....	728
expon .....	730
exprand .....	732
exprandi .....	734
expseg .....	736
expsega .....	738
expsegb .....	740
expsegba .....	742
expsegr .....	744
faustaudio .....	746
faustcompile .....	747
faustctl .....	748
faustgen .....	749
fareylen .....	751
fareyleni .....	753
ficlose .....	756
filebit .....	758
filelen .....	760
filenchnls .....	762
filepeak .....	764
filesca1 .....	766
filesr .....	768
filevalid .....	770
fillarray .....	772
fft .....	774
filter2 .....	776
fin .....	778
fini .....	780
fink .....	782
fiopen .....	784
flanger .....	786
flashtxt .....	788
FLbox .....	790
FLbutBank .....	795
FLbutton .....	798
FLcloseButton .....	803
FLcolor .....	806
FLcolor2 .....	808
FLcount .....	809
FLexecButton .....	812

---

FLgetsnap .....	815
FLgroup .....	816
FLgroupEnd .....	818
FLgroup_end .....	819
FLhide .....	820
FLhvsBox .....	821
FLhvsBoxSetValue .....	822
FLjoy .....	823
FLkeyIn .....	827
FLknob .....	829
FLlabel .....	834
FLloadsnap .....	836
FLmouse .....	837
flooper .....	839
flooper2 .....	841
floor .....	843
FLpack .....	845
FLpackEnd .....	848
FLpack_end .....	849
FLpanel .....	850
FLpanelEnd .....	854
FLpanel_end .....	855
FLprintk .....	856
FLprintk2 .....	857
FLroller .....	858
FLrun .....	861
FLsavesnap .....	862
FLscroll .....	868
FLscrollEnd .....	871
FLscroll_end .....	872
FLsetAlign .....	873
FLsetBox .....	874
FLsetColor .....	876
FLsetColor2 .....	878
FLsetFont .....	879
FLsetPosition .....	881
FLsetSize .....	882
FLsetsnap .....	883
FLsetSnapGroup .....	885
FLsetText .....	886
FLsetTextColor .....	888
FLsetTextSize .....	889
FLsetTextType .....	890
FLsetVal_i .....	893
FLsetVal .....	894
FLshow .....	895
FLslidBnk .....	896
FLslidBnk2 .....	900
FLslidBnkGetHandle .....	903
FLslidBnkSet .....	904
FLslidBnkSetk .....	905
FLslidBnk2Set .....	907
FLslidBnk2Setk .....	908
FLslider .....	911

---

FLtabs .....	917
FLtabsEnd .....	923
FLtabs_end .....	924
FLtext .....	925
FLupdate .....	928
fluidAllOut .....	929
fluidCCi .....	931
fluidCCk .....	933
fluidControl .....	935
fluidEngine .....	938
fluidLoad .....	941
fluidNote .....	943
fluidOut .....	945
fluidProgramSelect .....	948
fluidSetInterpMethod .....	951
FLvalue .....	953
FLvkeybd .....	956
FLvslidBnk .....	957
FLvslidBnk2 .....	961
FLxyin .....	963
fmanal .....	966
fmax .....	968
fmb3 .....	970
fmbell .....	972
fmin .....	975
fmmetal .....	977
fmod .....	979
fmpercfl .....	981
fmrhode .....	983
fmvoice .....	985
fmwurlie .....	987
fof .....	989
fof2 .....	992
fofilter .....	998
fog .....	1000
fold .....	1003
follow .....	1005
follow2 .....	1007
foscil .....	1009
foscili .....	1011
fout .....	1013
fouti .....	1017
foutir .....	1019
foutk .....	1021
fprintks .....	1023
fprints .....	1029
frac .....	1031
fractalnoise .....	1033
framebuffer .....	1035
freeverb .....	1037
ftchnls .....	1039
ftconv .....	1041
ftcps .....	1044
ftfree .....	1046

---

ftgen .....	1048
ftgenonce .....	1051
ftgentmp .....	1053
ftlen .....	1055
ftload .....	1057
ftloadk .....	1058
ftlptim .....	1059
ftmorf .....	1061
ftom .....	1063
ftsamplbank .....	1065
ftsave .....	1067
ftsavk .....	1069
ftsr .....	1070
gain .....	1072
gainslider .....	1074
gauss .....	1076
gaussi .....	1078
gausstrig .....	1080
gbuzz .....	1083
genarray .....	1085
genarray_i .....	1087
gendy .....	1089
gendyc .....	1093
gendyx .....	1096
getcfg .....	1100
getcol .....	1102
getftargs .....	1104
getrow .....	1106
getseed .....	1108
gogobel .....	1109
goto .....	1111
grain .....	1113
grain2 .....	1115
grain3 .....	1120
granule .....	1125
guiro .....	1128
harmon .....	1130
harmon2 .....	1132
hdf5read .....	1135
hdf5write .....	1137
hilbert .....	1139
hilbert2 .....	1144
hrtfearly .....	1146
hrtfmove .....	1150
hrtfmove2 .....	1153
hrtfverb .....	1156
hrtfstat .....	1159
hsboscil .....	1162
hvs1 .....	1165
hvs2 .....	1169
hvs3 .....	1175
hypot .....	1178
i .....	1180
if .....	1181

---

fftnv .....	1186
igoto .....	1188
ihold .....	1190
imagecreate .....	1192
imagefree .....	1194
imagegetpixel .....	1196
imageload .....	1199
imagesave .....	1201
imagesetpixel .....	1203
imagesize .....	1205
in .....	1207
in32 .....	1209
inch .....	1210
inh .....	1212
init .....	1213
initc14 .....	1216
initc21 .....	1217
initc7 .....	1218
inleta .....	1220
inletk .....	1223
inletkid .....	1225
inletf .....	1226
inletv .....	1227
ino .....	1228
inq .....	1229
inrg .....	1231
ins .....	1232
insremot .....	1234
insglobal .....	1237
instr .....	1239
int .....	1241
integ .....	1243
interp .....	1245
invalue .....	1248
inx .....	1250
inz .....	1251
JackoAudioIn .....	1252
JackoAudioInConnect .....	1253
JackoAudioOut .....	1254
JackoAudioOutConnect .....	1255
JackoFreewheel .....	1256
JackoInfo .....	1257
JackoInit .....	1259
JackoMidiInConnect .....	1261
JackoMidiOutConnect .....	1262
JackoMidiOut .....	1263
JackoNoteOut .....	1264
JackoOn .....	1265
JackoTransport .....	1266
jacktransport .....	1267
jitter .....	1269
jitter2 .....	1271
joystick .....	1273
jspline .....	1276

---

k	1278
K35_hpf	1279
K35_lpf	1283
kgoto	1287
kr	1289
ksmps	1290
lenarray	1291
lfo	1293
limit	1295
limit1	1297
line	1299
linen	1301
linenr	1303
lineto	1306
linlin	1308
link_beat_force	1310
link_beat_get	1311
link_beat_request	1312
link_create	1313
link_enable	1315
link_is_enabled	1317
link_metro	1319
link_peers	1321
link_tempo_get	1323
link_tempo_set	1325
linrand	1327
linseg	1329
linsegb	1331
linsegr	1333
liveconv	1335
locsend	1338
locsig	1341
log	1344
log10	1346
log2	1348
logbtwo	1350
logcurve	1352
loop_ge	1354
loop_gt	1356
loop_le	1358
loop_lt	1361
loopseg	1364
loopsegp	1366
looptseg	1368
loopxseg	1370
lorenz	1372
lorisread	1375
lorismorph	1378
lorisplay	1381
loscil	1384
loscil3	1387
loscilx	1390
lowpass2	1393
lowres	1395

---

lowresx .....	1397
lpf18 .....	1399
lpfreson .....	1401
lphasor .....	1403
lpinterp .....	1405
lposcil .....	1406
lposcil3 .....	1408
lposcila .....	1410
lposcilsa .....	1412
lposcilsa2 .....	1414
lpread .....	1416
lpreson .....	1419
lpshold .....	1422
lpsholdp .....	1424
lpslot .....	1425
lua_exec .....	1426
lua_opdef .....	1427
lua_opcall .....	1433
mac .....	1436
maca .....	1438
madsr .....	1440
mags .....	1444
mandel .....	1447
mandol .....	1450
maparray .....	1452
marimba .....	1455
massign .....	1458
max .....	1461
maxabs .....	1463
maxabsaccum .....	1465
maxaccum .....	1467
maxalloc .....	1469
max_k .....	1471
maxarray .....	1473
mclock .....	1475
mdelay .....	1477
median .....	1479
mediank .....	1481
metro .....	1483
mfb .....	1485
midglobal .....	1487
midiarp .....	1488
midic14 .....	1490
midic21 .....	1492
midic7 .....	1494
midichannelaftertouch .....	1496
midichn .....	1498
midicontrolchange .....	1501
midictrl .....	1503
mididefault .....	1505
midiin .....	1507
midifilestatus .....	1510
midinoteoff .....	1511
midinoteoncps .....	1513

---

midinoteonkey .....	1515
midinoteonoct .....	1517
midinoteonpch .....	1519
midion2 .....	1521
midion .....	1523
midiout .....	1526
midipitchbend .....	1528
midipolyaftertouch .....	1530
midiprogramchange .....	1533
miditempo .....	1535
midremot .....	1537
min .....	1540
minabs .....	1542
minabsaccum .....	1544
minaccum .....	1546
mincer .....	1548
minarray .....	1550
mirror .....	1552
MixerSetLevel .....	1554
MixerSetLevel_i .....	1557
MixerGetLevel .....	1558
MixerSend .....	1560
MixerReceive .....	1562
MixerClear .....	1564
mode .....	1566
modmatrix .....	1569
monitor .....	1574
moog .....	1576
moogladder .....	1578
moogladder2 .....	1580
moogvcf .....	1582
moogvcf2 .....	1584
moscil .....	1586
mp3in .....	1588
mp3len .....	1590
mp3scal .....	1592
mpulse .....	1594
mrtmsg .....	1596
mtof .....	1597
mton .....	1599
multitap .....	1601
mute .....	1603
mvchpf .....	1605
mvclpf1 .....	1607
mvclpf2 .....	1609
mvclpf3 .....	1611
mvclpf4 .....	1613
mxadsr .....	1615
nchnls .....	1618
nchnls_hw .....	1620
nchnls_i .....	1621
nestedap .....	1623
nlfilt .....	1626
nlfilt2 .....	1629

---

noise .....	1632
noteoff .....	1635
noteon .....	1636
noteondur2 .....	1637
noteondur .....	1639
notnum .....	1641
nreverb .....	1643
nrpn .....	1646
nsamp .....	1648
nstance .....	1650
ntom .....	1653
nstrnum .....	1655
ntrpol .....	1656
octave .....	1658
octcps .....	1660
octmidi .....	1663
octmidib .....	1665
octmidinn .....	1667
octpch .....	1670
olabuffer .....	1673
opcode .....	1675
oscbnk .....	1680
oscil1 .....	1686
oscil1i .....	1688
oscil3 .....	1690
oscil .....	1692
oscili .....	1694
oscilikt .....	1696
osciliktp .....	1698
oscilikts .....	1700
osciln .....	1702
oscils .....	1704
oscilx .....	1706
OSCinit .....	1707
OSClisten .....	1709
OSCrow .....	1713
OSCsend .....	1715
out32 .....	1717
out .....	1718
outc .....	1720
outch .....	1722
outh .....	1725
outiat .....	1726
outic14 .....	1728
outic .....	1729
outipat .....	1731
outipb .....	1732
outipc .....	1734
outkat .....	1736
outkc14 .....	1738
outkc .....	1739
outkpat .....	1741
outkpb .....	1742
outkpc .....	1744

---

outleta .....	1747
outletf .....	1749
outletk .....	1750
outletkid .....	1752
outletv .....	1753
outo .....	1754
outq1 .....	1755
outq2 .....	1757
outq3 .....	1759
outq4 .....	1761
outq .....	1763
outrg .....	1765
outs1 .....	1767
outs2 .....	1769
outs .....	1771
outvalue .....	1773
outx .....	1775
outz .....	1776
p5gconnect .....	1777
p5gdata .....	1779
p .....	1781
pan2 .....	1783
pan .....	1785
pareq .....	1787
partials .....	1790
partikkel .....	1792
partikkelget .....	1801
partikkelset .....	1804
partikkelsync .....	1807
passign .....	1811
paulstretch .....	1813
pcauchy .....	1815
pchbend .....	1817
pchmidi .....	1819
pchmidib .....	1821
pchmidinn .....	1823
pchoct .....	1826
pchtom .....	1829
pconvolve .....	1831
pcount .....	1834
pdclip .....	1837
pdhalf .....	1840
pdhalfy .....	1843
peak .....	1846
pgmassign .....	1848
phaser1 .....	1852
phaser2 .....	1855
phasor .....	1859
phasorbnk .....	1861
phs .....	1863
pindex .....	1866
pinker .....	1870
pinkish .....	1871
pitch .....	1874

---

pitchamdf .....	1877
planet .....	1880
platerrev .....	1883
plltrack .....	1885
pluck .....	1887
poisson .....	1889
pol2rect .....	1893
polyaft .....	1896
polynomial .....	1898
port .....	1901
portk .....	1903
poscil3 .....	1905
poscil .....	1908
pow .....	1910
powershape .....	1912
powoftwo .....	1914
prealloc .....	1916
prepiano .....	1918
print .....	1921
printf .....	1923
printk2 .....	1925
printk .....	1927
printks .....	1929
printks2 .....	1932
prints .....	1934
product .....	1936
product .....	1938
pset .....	1939
ptable .....	1941
ptablei .....	1943
ptable3 .....	1946
ptablew .....	1947
ptrack .....	1950
puts .....	1952
pvadd .....	1954
pvbufread .....	1958
pvcross .....	1960
pvinterp .....	1963
pvoc .....	1966
pvread .....	1968
pvsadsyn .....	1970
pvsanal .....	1972
pvsarp .....	1975
pvsbandp .....	1978
pvsbandr .....	1980
pvsbin .....	1982
pvsblur .....	1984
pvsbuffer .....	1986
pvsbufread .....	1987
pvsbufread2 .....	1990
pvscale .....	1992
pvscent .....	1994
pvsceps .....	1996
pvcross .....	1998

---

pvsdemix .....	2000
pvsdiskin .....	2002
pvsdisp .....	2004
pvsfilter .....	2006
pvsfread .....	2009
pvsfreeze .....	2011
pvsftr .....	2013
pvsftw .....	2015
pvsfwrite .....	2017
pvsgain .....	2019
pvshift .....	2021
pvsifd .....	2023
pvsinfo .....	2025
pvsinit .....	2027
pvsin .....	2028
pvslock .....	2029
pvsmaska .....	2031
pvsmix .....	2033
pvsmorph .....	2035
pvssmooth .....	2038
pvsout .....	2040
pvsosc .....	2041
pvspitch .....	2044
pvstanal .....	2047
pvstencil .....	2049
pvstrace .....	2051
pvsvoc .....	2053
pvsynth .....	2055
pvswarp .....	2057
pvs2tab .....	2059
pyassign Opcodes .....	2060
pycall Opcodes .....	2061
pyeval Opcodes .....	2065
pyexec Opcodes .....	2066
pyinit Opcodes .....	2069
pyrun Opcodes .....	2070
pwd .....	2072
qinf .....	2074
qnan .....	2076
r2c .....	2078
rand .....	2080
randh .....	2082
randi .....	2084
random .....	2086
randomh .....	2088
randomi .....	2091
rbjeq .....	2094
readclock .....	2097
readf .....	2099
readfi .....	2101
readk .....	2103
readk2 .....	2106
readk3 .....	2109
readk4 .....	2112

---

readscore .....	2115
readscratch .....	2117
rect2pol .....	2119
reinit .....	2121
release .....	2123
remoteport .....	2124
remove .....	2125
repluck .....	2126
reson .....	2128
resonk .....	2130
resonr .....	2132
resonx .....	2135
resonxk .....	2137
resony .....	2139
resonz .....	2141
resyn .....	2144
return .....	2146
reverb .....	2147
reverb2 .....	2149
reverb3c .....	2150
rewindscore .....	2152
rezzy .....	2154
rfft .....	2156
rifft .....	2158
rigoto .....	2160
rireturn .....	2161
rms .....	2163
rnd .....	2165
rnd31 .....	2167
round .....	2173
rspline .....	2175
rtclock .....	2177
<b>S</b> .....	2179
s16b14 .....	2180
s32b14 .....	2182
samphold .....	2184
sandpaper .....	2186
scale .....	2188
scalearray .....	2190
scanhammer .....	2192
scans .....	2193
scantable .....	2197
scanu .....	2199
schedkwhen .....	2201
schedkwhennamed .....	2204
schedule .....	2206
schedwhen .....	2209
scoreline .....	2211
scoreline_i .....	2213
sc_lag .....	2215
sc_lagud .....	2217
sc_phasor .....	2219
sc_trig .....	2221
seed .....	2223

---

sekere .....	2225
select .....	2227
semitone .....	2229
sense .....	2231
sensekey .....	2232
serialBegin .....	2236
serialEnd .....	2238
serialFlush .....	2239
serialPrint .....	2240
serialRead .....	2241
serialWrite_i .....	2243
serialWrite .....	2244
seqtime2 .....	2246
seqtime .....	2249
setctrl .....	2252
setksmps .....	2255
setrow .....	2257
setscorepos .....	2259
sfilist .....	2261
sfinstr3 .....	2263
sfinstr3m .....	2266
sfinstr .....	2269
sfinstrm .....	2272
sfloat .....	2274
sflooper .....	2277
sfpassign .....	2280
sfplay3 .....	2283
sfplay3m .....	2286
sfplay .....	2289
sfplaym .....	2291
sfplist .....	2294
sfpreset .....	2296
shaker .....	2299
shiftin .....	2301
shiftout .....	2303
signum .....	2305
sin .....	2307
sinh .....	2309
sininv .....	2311
sinsyn .....	2313
sleighbells .....	2315
slicearray .....	2317
slider16 .....	2319
slider16f .....	2321
slider16table .....	2323
slider16tablef .....	2325
slider32 .....	2327
slider32f .....	2329
slider32table .....	2331
slider32tablef .....	2333
slider64 .....	2335
slider64f .....	2337
slider64table .....	2339
slider64tablef .....	2341

---

slider8 .....	2343
slider8f .....	2345
slider8table .....	2347
slider8tablef .....	2349
sliderKawai .....	2351
sndloop .....	2352
sndwarp .....	2354
sndwarpst .....	2358
sockrecv .....	2362
socksend .....	2364
sorta .....	2366
sortd .....	2367
soundin .....	2368
space .....	2371
spat3d .....	2376
spat3di .....	2385
spat3dt .....	2389
spdist .....	2394
specaddm .....	2398
specdiff .....	2399
specdisp .....	2400
specfilt .....	2401
spechist .....	2402
specptrk .....	2403
specscal .....	2405
specsum .....	2406
spectrum .....	2407
splitrig .....	2409
sprintf .....	2411
sprintfk .....	2413
spsend .....	2415
sqrt .....	2418
sr .....	2420
statevar .....	2422
stix .....	2424
STKBandedWG .....	2426
STKBeeThree .....	2428
STKBlowBotl .....	2430
STKBlowHole .....	2432
STKBowed .....	2434
STKBrass .....	2436
STKClarinet .....	2438
STKDrummer .....	2440
STKFlute .....	2442
STKFMVoices .....	2444
STKHevyMetl .....	2446
STKMandolin .....	2448
STKModalBar .....	2450
STKMoog .....	2452
STKPercFlut .....	2454
STKPlucked .....	2456
STKResonate .....	2458
STKRhodey .....	2460
STKSaxofony .....	2462

---

STKShakers .....	2464
STKSimple .....	2466
STKSitar .....	2468
STKStifKarp .....	2470
STKTubeBell .....	2472
STKVoicForm .....	2474
STKWhistle .....	2476
STKWurley .....	2478
strchar .....	2480
strchark .....	2482
strcpy .....	2483
strcpyk .....	2484
strcat .....	2486
strcatk .....	2488
strcmp .....	2490
strcmpk .....	2492
streson .....	2493
strfromurl .....	2495
strget .....	2497
strindex .....	2499
strindexk .....	2500
strlen .....	2502
strlenk .....	2503
strlower .....	2504
strlowerk .....	2506
strrindex .....	2507
strrindexk .....	2509
strset .....	2510
strsub .....	2512
strsubk .....	2514
strtod .....	2515
strtodk .....	2516
strtol .....	2517
strtolk .....	2518
strupper .....	2519
strupperk .....	2520
subinstr .....	2521
subinstrinit .....	2524
sum .....	2525
sumarray .....	2527
svfilter .....	2529
syncgrain .....	2532
syncloop .....	2535
syncphasor .....	2537
system .....	2541
tb .....	2543
tab .....	2546
tabifd .....	2548
table .....	2550
table3 .....	2552
tablecopy .....	2553
tablefilter .....	2556
tablefilteri .....	2558
tablegpw .....	2560

---

tablei .....	2561
tableicopy .....	2564
tableigpw .....	2565
tableikt .....	2566
tableimix .....	2569
tableiw .....	2571
tablekt .....	2575
tablemix .....	2578
tableng .....	2580
tablera .....	2582
tableseg .....	2585
tableshuffle .....	2587
tablew .....	2590
tablewa .....	2593
tablewkt .....	2596
tablexkt .....	2598
tablexseg .....	2601
tabmorph .....	2603
tabmorpha .....	2606
tabmorphak .....	2608
tabmorphi .....	2611
tabplay .....	2614
tabrec .....	2615
tabsum .....	2616
tab2pvs .....	2618
tambourine .....	2619
tan .....	2621
tanh .....	2623
taninv .....	2625
taninv2 .....	2627
tbvcf .....	2629
tempest .....	2632
tempo .....	2635
temposcal .....	2637
tempoval .....	2639
tigoto .....	2641
timedseq .....	2643
timeinstk .....	2646
timeinsts .....	2648
timek .....	2650
times .....	2652
timeout .....	2655
tival .....	2657
tlineto .....	2659
tone .....	2661
tonek .....	2663
tonex .....	2665
trandom .....	2667
tradsyn .....	2669
transeg .....	2671
transegb .....	2673
transegr .....	2675
trcross .....	2677
trfilter .....	2679

---

trhighest .....	2681
trigger .....	2683
trigseq .....	2685
trirand .....	2689
trlowest .....	2691
trmix .....	2693
trscale .....	2695
trshift .....	2697
trsplitt .....	2699
turnoff .....	2701
turnoff2 .....	2703
turnon .....	2706
tvconv .....	2707
unirand .....	2709
until .....	2711
unwrap .....	2713
upsamp .....	2716
urandom .....	2718
urd .....	2721
vactrol .....	2724
vadd .....	2726
vadd_i .....	2729
vaddv .....	2731
vaddv_i .....	2734
vaget .....	2736
valpass .....	2739
vaset .....	2742
vbap .....	2744
vbapmove .....	2747
vbapg .....	2750
vbapgmove .....	2753
vbap16 .....	2756
vbap16move .....	2758
vbap4 .....	2760
vbap4move .....	2763
vbap8 .....	2766
vbap8move .....	2768
vbaplsinit .....	2771
vbapz .....	2774
vbapzmove .....	2776
vcella .....	2778
vco .....	2781
vco2 .....	2784
vco2ft .....	2788
vco2ift .....	2790
vco2init .....	2792
vcomb .....	2795
vcopy .....	2798
vcopy_i .....	2801
vdelay .....	2803
vdelay3 .....	2805
vdelayx .....	2807
vdelayxq .....	2809
vdelayxs .....	2811

---

vdelayxw .....	2813
vdelayxwq .....	2815
vdelayxws .....	2817
vdelayk .....	2819
vdivv .....	2820
vdivv_i .....	2823
vecdelay .....	2825
veloc .....	2826
vexp .....	2828
vexp_i .....	2831
vexpseg .....	2833
vexpv .....	2835
vexpv_i .....	2838
vibes .....	2840
vibr .....	2842
vibrato .....	2844
vincr .....	2846
vlimit .....	2849
vlinseg .....	2850
vlowres .....	2852
vmap .....	2854
vmirror .....	2856
vmult .....	2857
vmult_i .....	2861
vmultv .....	2863
vmultv_i .....	2866
voice .....	2868
vosim .....	2871
vphaseseg .....	2876
vport .....	2878
vpow .....	2879
vpow_i .....	2883
vpowv .....	2886
vpowv_i .....	2889
vpvoc .....	2891
vrandh .....	2894
vrandi .....	2897
vstaudio, vstaudiog .....	2900
vstbankload .....	2902
vstedit .....	2903
vstinit .....	2905
vstinfo .....	2907
vstmidiout .....	2909
vstnote .....	2911
vstparamset, vstparamget .....	2913
vstprogset .....	2915
vsubv .....	2916
vsubv_i .....	2919
vtable1k .....	2921
vtablei .....	2923
vtablek .....	2925
vtablea .....	2927
vtablewi .....	2929
vtablewk .....	2930

---

vtablewa .....	2932
vtabi .....	2934
vtabk .....	2936
vtaba .....	2938
vtabwi .....	2940
vtabwk .....	2941
vtabwa .....	2942
vwrap .....	2943
waveset .....	2944
websocket .....	2946
weibull .....	2948
wgbow .....	2951
wgbowedbar .....	2953
wgbrass .....	2955
wgclar .....	2957
wgflute .....	2959
wgpluck .....	2961
wgpluck2 .....	2964
wguide1 .....	2966
wguide2 .....	2969
while .....	2972
wiiconnect .....	2974
wiidata .....	2976
wiirange .....	2979
wiisend .....	2980
window .....	2982
wrap .....	2985
writescratch .....	2987
wterrain .....	2989
xadsr .....	2991
xin .....	2993
xout .....	2995
xscanmap .....	2997
xscansmap .....	3000
xscans .....	3001
xscanu .....	3005
xtratim .....	3009
xyin .....	3013
xyscale .....	3015
zacl .....	3017
zakinit .....	3019
zamod .....	3022
zar .....	3024
zarg .....	3026
zaw .....	3028
zawm .....	3030
zdf_1pole .....	3033
zdf_1pole_mode .....	3035
zdf_2pole .....	3037
zdf_2pole_mode .....	3039
zdf_ladder .....	3041
zfilter2 .....	3043
zir .....	3045
ziw .....	3047

ziwm .....	3049
zkcl .....	3051
zkmod .....	3053
zkr .....	3055
zkw .....	3057
zkwm .....	3059
Score Statements and GEN Routines .....	3062
Score Statements .....	3062
a Statement (or Advance Statement) .....	3063
b Statement .....	3065
C Statement .....	3067
d Statement (De Note Statement) .....	3069
e Statement .....	3071
f Statement (or Function Table Statement) .....	3073
i Statement (Instrument or Note Statement) .....	3075
m Statement (Mark Statement) .....	3079
n Statement .....	3081
q Statement .....	3083
r Statement (Repeat Statement) .....	3085
s Statement .....	3087
t Statement (Tempo Statement) .....	3089
v Statement .....	3091
x Statement .....	3093
y Statement (or Seed Statement) .....	3095
{ Statement .....	3097
} Statement .....	3100
GEN Routines .....	3100
GEN01 .....	3104
GEN02 .....	3107
GEN03 .....	3109
GEN04 .....	3112
GEN05 .....	3115
GEN06 .....	3117
GEN07 .....	3119
GEN08 .....	3121
GEN09 .....	3123
GEN10 .....	3126
GEN11 .....	3128
GEN12 .....	3130
GEN13 .....	3133
GEN14 .....	3137
GEN15 .....	3140
GEN16 .....	3148
GEN17 .....	3151
GEN18 .....	3153
GEN19 .....	3156
GEN20 .....	3158
GEN21 .....	3161
GEN23 .....	3165
GEN24 .....	3167
GEN25 .....	3169
GEN27 .....	3171
GEN28 .....	3173
GEN30 .....	3176

GEN31 .....	3178
GEN32 .....	3179
GEN33 .....	3181
GEN34 .....	3184
GEN40 .....	3187
GEN41 .....	3189
GEN42 .....	3191
GEN43 .....	3193
GEN49 .....	3194
GEN51 .....	3196
GEN52 .....	3199
GENtanh .....	3202
GENexp .....	3204
GENsone .....	3206
GENquadbezier .....	3208
GENfarey .....	3211
GENwave .....	3216
GENpadsynth .....	3219
Experimental Orchestra Opcodes and GEN Routines .....	3223
Experimental Orchestra Opcodes .....	3223
cudanal .....	3224
cudasynth .....	3227
cudasliding .....	3229
Deprecated Orchestra Opcodes and GEN Routines .....	3231
Deprecated Orchestra Opcodes .....	3231
abetarand .....	3232
abexprnd .....	3233
acauchy .....	3234
aexprand .....	3235
agauss .....	3236
agogobel .....	3237
alinrand .....	3238
apcauchy .....	3239
apoisson .....	3240
apow .....	3241
array .....	3242
atrirand .....	3244
aunirand .....	3245
aweibull .....	3246
bformdec .....	3247
bformenc .....	3249
clock .....	3251
hrtfer .....	3252
ibetarand .....	3254
ibexprnd .....	3255
icauchy .....	3256
ictrl14 .....	3257
ictrl21 .....	3258
ictrl7 .....	3259
iexprand .....	3260
igauss .....	3261
ilinrand .....	3262
imidic14 .....	3263
imidic21 .....	3264

---

imidic7 .....	3265
instimek .....	3266
instimes .....	3267
ioff .....	3268
ion .....	3269
iondur2 .....	3270
iondur .....	3271
ioutat .....	3272
ioutc14 .....	3273
ioutc .....	3274
ioutpat .....	3275
ioutpb .....	3276
ioutpc .....	3277
ipcauchy .....	3278
ipoisson .....	3279
ipow .....	3280
is16b14 .....	3281
is32b14 .....	3282
islider16 .....	3283
islider32 .....	3284
islider64 .....	3285
islider8 .....	3286
itablecopy .....	3287
itablegpw .....	3288
itablemix .....	3289
itablew .....	3290
itrirand .....	3291
iunirand .....	3292
iweibull .....	3293
kbetarand .....	3294
kbexprnd .....	3295
kcauchy .....	3296
kdump2 .....	3297
kdump3 .....	3298
kdump4 .....	3299
kdump .....	3300
kexprand .....	3301
kfilter2 .....	3302
kgauss .....	3303
klinrand .....	3304
kon .....	3305
koutat .....	3306
koutc14 .....	3307
koutc .....	3308
koutpat .....	3309
koutpb .....	3310
koutpc .....	3311
kpcachy .....	3312
kpoisson .....	3313
kpow .....	3314
kread2 .....	3315
kread3 .....	3316
kread4 .....	3317
kread .....	3318

ktableseg .....	3319
ktrirand .....	3320
kunirand .....	3321
kweibull .....	3322
sndload .....	3323
peakk .....	3325
pop .....	3326
pop_f .....	3328
push .....	3329
push_f .....	3331
soundout .....	3332
soundouts .....	3334
stack .....	3335
Deprecated Gen Routines .....	3336
GEN22 .....	3337
The Utility Programs .....	3338
Directories .....	3338
Soundfile Formats .....	3338
Analysis File Generation (ATSA, CVANAL, HETRO, LPANAL, PVANAL) .....	3339
File Queries (SNDINFO) .....	3349
File Conversion (HET_IMPORT, HET_EXPORT, PVLOOK, PV_EXPORT, PV_IMPORT, SDIF2AD, SRCONV) .....	3350
Other Csound Utilities (CS, CSB64ENC, ENVEXT, EXTRACTOR, MAKECSD, MIXER, SCALE, MKDB) .....	3367
Cscore .....	3381
Events, Lists, and Operations .....	3381
Writing a Cscore Control Program .....	3384
Compiling a Cscore Program .....	3388
More Advanced Examples .....	3391
Csbeats .....	3393

---

# Orchestra Opcodes and Operators

# !=

!= — Determines if one value is not equal to another.

## Description

Determines if one value is not equal to another.

## Syntax

```
( a != b ? v1 : v2 )
```

where  $a$ ,  $b$ ,  $v1$  and  $v2$  may be expressions, but  $a$ ,  $b$  not audio-rate.

## Performance

In the above conditional,  $a$  and  $b$  are first compared. If the indicated relation is true ( $a$  not equal to  $b$ ), then the conditional expression has the value of  $v1$ ; if the relation is false, the expression has the value of  $v2$ .

NB.: If  $v1$  or  $v2$  are expressions, these will be evaluated before the conditional is determined.

In terms of binding strength, all conditional operators (i.e. the relational operators (<, etc.), and ?, and :) are weaker than the arithmetic and logical operators (+, -, \*, /, && and //).

These are *operators* not *opcodes*. Therefore, they can be used within orchestra statements, but do not form complete statements themselves.

## Examples

Here is an example of the != operator. It uses the file *notequal.csd* [examples/notequal.csd].

### Example 13. Example of the != operator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o notequal.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ipch = cpspch(p4)
```

```
iprint = p5
if (iprint != 1) goto skipPrint
print ipch
asig vco .7, ipch, 2, 0.5, 1
outs asig, asig

skipPrint:

endin

</CsInstruments>
<CsScore>
f 1 0 65536 10 1 ;sine wave

i1 0 .5 8.00 0
i1 + .5 8.01 1 ; this note will print it's ipch value and only this one will be played
i1 + .5 8.02 2

e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
instr 1: ipch = 277.167
```

## See Also

`==, >=, >, <=, <`

# #define

#define — Defines a macro.

## Description

Macros are textual replacements which are made in the orchestra as it is being read. The macro system in Csound is a very simple one, and uses the characters # and \$ to define and call macros. This can save typing, and can lead to a coherent structure and consistent style. This is similar to, but independent of, the *macro system in the score language*.

*#define NAME* -- defines a simple macro. The name of the macro must begin with a letter and can consist of any combination of letters and numbers. Case is significant. This form is limiting, in that the variable names are fixed. More flexibility can be obtained by using a macro with arguments, described below.

*#define NAME(a' b' c')* -- defines a macro with arguments. This can be used in more complex situations. The name of the macro must begin with a letter and can consist of any combination of letters and numbers. Within the replacement text, the arguments can be substituted by the form: \$A. In fact, the implementation defines the arguments as simple macros. The names may be any choice of letters. Remember that case is significant in macro names.



### Note

In earlier versions of Csound there was a limit of 5 arguments, but this restriction was removed in 2011; memory is the only constraint.

## Syntax

```
#define NAME # replacement text #  
#define NAME(a' b' c') # replacement text #
```

## Initialization

*# replacement text #* -- The replacement text is any character string (not containing a #) and can extend over multiple lines. The replacement text is enclosed within the # characters, which ensure that additional characters are not inadvertently captured.

## Performance

Some care is needed with textual replacement macros, as they can sometimes do strange things. They take no notice of any meaning, so spaces are significant. This is why, unlike the C programming language, the definition has the replacement text surrounded by # characters. Used carefully, this simple macro system is a powerful concept, but it can be abused.

## Examples

Here is a simple example of the defining a macro. It uses the file *define.csd* [examples/define.csd].

### Example 14. Simple example of the define macro.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o define.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Define the macros.
#define VOLUME #5000#
#define FREQ #440#
#define TABLE #1#

; Instrument #1
instr 1
; Use the macros.
; This will be expanded to "a1 oscil 5000, 440, 1".
a1 oscil $VOLUME, $FREQ, $TABLE

; Send it to the output.
out a1
endin

</CsInstruments>
<CsScore>

; Define Table #1 with an ordinary sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e

</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like this:

```

Macro definition for VOLUME
Macro definition for CPS
Macro definition for TABLE

```

Here is an example of the defining a macro with arguments. It uses the file *define\_args.csd* [examples/define\_args.csd].

### Example 15. Example of the define macro with arguments.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages

```

```

-odac          -iadc      -d      ;;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o define_args.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Define the oscillator macro.
#define OSCMACRO(VOLUME'FREQ'TABLE) #oscil $VOLUME, $FREQ, $TABLE#

; Instrument #1
instr 1
; Use the oscillator macro.
; This will be expanded to "a1 oscil 5000, 440, 1".
a1 $OSCMACRO(5000'440'1)

; Send it to the output.
out a1
endin

</CsInstruments>
<CsScore>

; Define Table #1 with an ordinary sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e

</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like this:

Macro definition for OSCMACRO

## Predefined Math Constant Macros

New in Csound 5.04 are predefined Math Constant Macros. The values defined are those found in the C header math.h, and are automatically defined when Csound starts and available for use in orchestras.

Macro	Value	Equivalent to
\$M_E	2.7182818284590452354	e
\$M_LOG2E	1.4426950408889634074	log <sub>2</sub> (e)
\$M_LOG10E	0.43429448190325182765	log <sub>10</sub> (e)
\$M_LN2	0.69314718055994530942	log <sub>e</sub> (2)
\$M_LN10	2.30258509299404568402	log <sub>e</sub> (10)
\$M_PI	3.14159265358979323846	pi
\$M_PI_2	1.57079632679489661923	pi/2
\$M_PI_4	0.78539816339744830962	pi/4

Macro	Value	Equivalent to
\$M_1_PI	0.31830988618379067154	1/pi
\$M_2_PI	0.63661977236758134308	2/pi
\$M_2_SQRTPI	1.12837916709551257390	2/sqrt(pi)
\$M_SQRT2	1.41421356237309504880	sqrt(2)
\$M_SQRT1_2	0.70710678118654752440	1/sqrt(2)

## See Also

*\$NAME, #undef*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
April 1998

Examples written by Kevin Conder.

New in Csound version 3.48

# #include

#include — Includes an external file for processing.

## Description

Includes an external file for processing.

## Syntax

```
#include "filename"
```

## Performance

It is sometimes convenient to have the orchestra arranged in a number of files, for example with each instrument in a separate file. This style is supported by the *#include* facility which is part of the macro system. A line containing the text

```
#include "filename"
```

where the character " can be replaced by any suitable character. For most uses the double quote symbol will probably be the most convenient. The file name can include a full path.

This takes input from the named file until it ends, when input reverts to the previous input. *Note:* Csound versions prior to 4.19 had a limit of 20 on the depth of included files and macros.

Another suggested use of *#include* would be to define a set of macros which are part of the composer's style.

An extreme form would be to have each instrument defines as a macro, with the instrument number as a parameter. Then an entire orchestra could be constructed from a number of *#include* statements followed by macro calls.

```
#include "clarinet"  
#include "flute"  
#include "bassoon"  
$CLARINET(1)  
$FLUTE(2)  
$BASSOON(3)
```

It must be stressed that these changes are at the textual level and so take no cognizance of any meaning.

If the version of Csound is built with the CURL library the filename in an *#include* statement can be an URL, recognised by including the substring "://" in the name. This will include text via protocols such as http, https, and ftp.

## Examples

Here is an example of the include opcode. It uses the file *include.csd* [examples/include.csd], and *table1.inc* [examples/table1.inc].

**Example 16. Example of the include opcode.**

```
/* table1.inc */
; Table #1, a sine wave.
f 1 0 16384 10 1
/* table1.inc */
```

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      -iadc      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o include.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - a basic oscillator.
instr 1
  kamp = 10000
  kcps = 440
  ifn = 1

  a1 oscil kamp, kcps, ifn
  out a1
endin

</CsInstruments>
<CsScore>

; Include the file for Table #1.
#include "table1.inc"

; Play Instrument #1 for 2 seconds.
i 1 0 2
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
April 1998

Example written by Kevin Conder.

New in Csound version 3.48

URLs available from Csound 6.02

# #undef

#undef — Un-defines a macro.

## Description

Macros are textual replacements which are made in the orchestra as it is being read. The macro system in Csound is a very simple one, and uses the characters # and \$ to define and call macros. This can save typing, and can lead to a coherent structure and consistent style. This is similar to, but independent of, the *macro system in the score language*.

*#undef NAME* -- undefines a macro name. If a macro is no longer required, it can be undefined with *#undef NAME*.

## Syntax

**#undef** NAME

## Performance

Some care is needed with textual replacement macros, as they can sometimes do strange things. They take no notice of any meaning, so spaces are significant. This is why, unlike the C programming language, the definition has the replacement text surrounded by # characters. Used carefully, this simple macro system is a powerful concept, but it can be abused.

## See Also

*#define*, *\$NAME*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
April 1998

New in Csound version 3.48

## #ifdef

`#ifdef` — Conditional reading of code.

### Description

If a macro is defined then `#ifdef` can incorporate text into an orchestra upto the next `#end`. This is similar to, but independent of, the *macro system in the score language*.

### Syntax

```
#ifdef NAME
....
#else
....
#end
```

### Performance

Note that the `#ifdef` can be nested, like in the C preprocessor language.

### Examples

Here is a simple example of the conditional.

#### Example 17. Simple example of the `#ifdef` form.

```
#define debug ##
  instr 1
#ifdef debug
  print "calling oscil"
#end
a1  oscil 32000,440,1
out a1
endin
```

### See Also

`$NAME`, `#define`, `#ifndef`.

### Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
April 2005

New in Csound5 (and 4.23f13)

## #ifndef

`#ifndef` — Conditional reading of code.

### Description

If the specified macro is not defined then `#ifndef` can incorporate text into an orchestra upto the next `#end`. This is similar to, but independent of, the *macro system in the score language*.

### Syntax

```
#ifndef NAME  
....  
#else  
....  
#end
```

### Performance

Note that the `#ifndef` can be nested, like in the C preprocessor language.

### Examples

Here is a simple example of the conditional.

#### Example 18. Simple example of the `#ifndef` form.

```
define final ##  
instr 1  
ifndef final  
print "calling oscil"  
end  
a1 oscil 32000,440,1  
out a1  
endin
```

### See Also

`$NAME`, `#define`, `#ifdef`.

### Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
April 2005

New in Csound5 (and 4.23f13)

# \$NAME

\$NAME — Calls a defined macro.

## Description

Macros are textual replacements which are made in the orchestra as it is being read. The macro system in Csound is a very simple one, and uses the characters # and \$ to define and call macros. This can save typing, and can lead to a coherent structure and consistent style. This is similar to, but independent of, the *macro system in the score language*.

*\$NAME* -- calls a defined macro. To use a macro, the name is used following a \$ character. The name is terminated by the first character which is neither a letter nor a number. If it is necessary for the name not to terminate with a space, a period, which will be ignored, can be used to terminate the name. The string, *\$NAME.*, is replaced by the replacement text from the definition. The replacement text can also include macro calls.

## Syntax

**\$NAME**

## Initialization

*# replacement text #* -- The replacement text is any character string (not containing a #) and can extend over multiple lines. The replacement text is enclosed within the # characters, which ensure that additional characters are not inadvertently captured.

## Performance

Some care is needed with textual replacement macros, as they can sometimes do strange things. They take no notice of any meaning, so spaces are significant. This is why, unlike the C programming language, the definition has the replacement text surrounded by # characters. Used carefully, this simple macro system is a powerful concept, but it can be abused.

## Examples

Here is an example of the calling a macro. It uses the file *define.csd* [examples/define.csd].

### Example 19. An example of the calling a macro.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc    -d      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o define.wav -W ;; for file output any platform
</CsOptions>
```

```

<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Define the macros.
#define VOLUME #5000#
#define FREQ #440#
#define TABLE #1#

; Instrument #1
instr 1
; Use the macros.
; This will be expanded to "a1 oscil 5000, 440, 1".
a1 oscil $VOLUME, $FREQ, $TABLE

; Send it to the output.
out a1
endin

</CsInstruments>
<CsScore>

; Define Table #1 with an ordinary sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e

</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like this:

```

Macro definition for VOLUME
Macro definition for CPS
Macro definition for TABLE

```

Here is an example of the calling a macro with arguments. It uses the file *define\_args.csd* [examples/define\_args.csd].

### Example 20. An example of the calling a macro with arguments.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc    -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o define_args.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10

```

```
nchnls = 1

; Define the oscillator macro.
#define OSCMACRO(VOLUME'FREQ'TABLE) #oscil $VOLUME, $FREQ, $TABLE#

; Instrument #1
instr 1
; Use the oscillator macro.
; This will be expanded to "a1 oscil 5000, 440, 1".
a1 $OSCMACRO(5000'440'1)

; Send it to the output.
out a1
endin

</CsInstruments>
<CsScore>

; Define Table #1 with an ordinary sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include a line like this:

```
Macro definition for OSCMACRO
```

## See Also

*#define, #undef*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
April, 1998

Examples written by Kevin Conder.

New in Csound version 3.48

# %

% — Modulus operator.

## Description

Arithmetic operators perform operations of change-sign (negate), don't-change-sign, logical AND logical OR, add, subtract, multiply and divide. Note that a value or an expression may fall between two of these operators, either of which could take it as its left or right argument, as in

$a + b * c$ .

In such cases three rules apply:

1.  $*$  and  $/$  bind to their neighbors more strongly than  $+$  and  $_$ . Thus the above expression is taken as

$a + (b * c)$

with  $*$  taking  $b$  and  $c$  and then  $+$  taking  $a$  and  $b * c$ .

2.  $+$  and  $-$  bind more strongly than  $\&\&$ , which in turn is stronger than  $\|\|$ :

$a \&\& b - c \|\| d$

is taken as

$(a \&\& (b - c)) \|\| d$

3. When both operators bind equally strongly, the operations are done left to right:

$a - b - c$

is taken as

$(a - b) - c$

Parentheses may be used as above to force particular groupings.

The operator  $\%$  returns the value of  $a$  reduced by  $b$ , so that the result, in absolute value, is less than the absolute value of  $b$ , by repeated subtraction. This is the same as modulus function in integers. New in Csound version 3.50.

## Syntax

$a \% b$  (no rate restriction)

where the arguments  $a$  and  $b$  may be further expressions.

## Arguments

The arguments of  $\%$  can be scalar values or k-rate one dimensional arrays (vectors), or any combination. If one of the arguments is an array, so is the value.

## Examples

Here is an example of the % operator. It uses the file *modulus.csd* [examples/modulus.csd].

### Example 21. Example of the % operator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -Ma --midi-key=4 --midi-velocity-amp=5 -m0 ;;realtime audio out and midi in
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o %.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giScale ftgen 1, 0, -12, -2, 0, -34, -7, 10, -14, 3, -21, -3, -27, -10, 7, -17 ;12 note scale with detu

instr 1

ikey = p4
ivel = p5
indx = ikey % 12 ;work on the twelftone scale
icent tab_i indx, giScale ;load the scale
ifreqeq = cpsmidinn(ikey)
ifreq = ifreqeq * cent(icent) ;change frequency by cents from table
prints "Key %d modulus 12 = %d. ", ikey, indx
prints "Equal-tempered frequency of this key = %f," , ifreqeq
prints " but here with cent deviation %d = %f%n", icent, ifreq
asig vco2 ivel*.5, ifreq
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 0 60 ;run for 60 seconds

e
</CsScore>
</CsoundSynthesizer>

```

## See Also

-, +, &&, //, \*, /, ^

## Credits

Arithmetic on vectors is new in version 6.00

# &&

&& — Logical AND operator.

## Description

Arithmetic operators perform operations of change-sign (negate), don't-change-sign, logical AND logical OR, add, subtract, multiply and divide. Note that a value or an expression may fall between two of these operators, either of which could take it as its left or right argument, as in

$a + b * c$ .

In such cases three rules apply:

1. \* and / bind to their neighbors more strongly than + and -. Thus the above expression is taken as

$a + (b * c)$

with \* taking b and c and then + taking a and b \* c.

2. + and # bind more strongly than &&, which in turn is stronger than ||:

$a \&\& b - c \parallel d$

is taken as

$(a \&\& (b - c)) \parallel d$

3. When both operators bind equally strongly, the operations are done left to right:

$a - b - c$

is taken as

$(a - b) - c$

Parentheses may be used as above to force particular groupings.

## Syntax

`a && b` (logical AND; not audio-rate)

where the arguments *a* and *b* may be further expressions.

## Examples

Here is an example of the Logical AND operator. It uses the file *opand.csd* [examples/opand.csd].

### Example 22. Example of the opand opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o opand.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kval  randomh 0, 1.2, 20 ;choose between 0 and 1.2
if kval >0 && kval<=.5 then ;3 possible outcomes
  kval = 1
elseif kval >.5 && kval<=1 then
  kval =2
elseif kval >1 then
  kval =3
endif

printk2 kval ;print value when it changes
asig  poscil .7, 440*kval, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
f1 0 16384 10 1

i 1 0 5
e
</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like these:

```

i1      0.00000
i1      3.00000
i1      1.00000
i1      3.00000
i1      1.00000
i1      2.00000
i1      3.00000
.....

```

## See Also

-, +, //, \*, /, ^, %

## >

> — Determines if one value is greater than another.

## Description

Determines if one value is greater than another.

## Syntax

```
( a > b ? v1 : v2 )
```

where  $a$ ,  $b$ ,  $v1$  and  $v2$  may be expressions, but  $a$ ,  $b$  not audio-rate.

## Performance

In the above conditional,  $a$  and  $b$  are first compared. If the indicated relation is true ( $a$  greater than  $b$ ), then the conditional expression has the value of  $v1$ ; if the relation is false, the expression has the value of  $v2$ .

NB.: If  $v1$  or  $v2$  are expressions, these will be evaluated before the conditional is determined.

In terms of binding strength, all conditional operators (i.e. the relational operators (<, etc.), and ?, and :) are weaker than the arithmetic and logical operators (+, -, \*, /, && and //).

These are *operators* not *opcodes*. Therefore, they can be used within orchestra statements, but do not form complete statements themselves.

## Examples

Here is an example of the > operator. It uses the file *greaterthan.csd* [examples/greaterthan.csd].

### Example 23. Example of the > operator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o greaterthan.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1.
instr 1
```

```
; Get the 4th p-field from the score.
k1 = p4

; Is it greater than 3? (1 = true, 0 = false)
k2 = (p4 > 3 ? 1 : 0)

; Print the values of k1 and k2.
printks "k1 = %f, k2 = %f\\n", 1, k1, k2
endin

</CsInstruments>
<CsScore>

; Call Instrument #1 with a p4 = 2.
i 1 0 0.5 2
; Call Instrument #1 with a p4 = 3.
i 1 1 0.5 3
; Call Instrument #1 with a p4 = 4.
i 1 2 0.5 4
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
k1 = 2.000000, k2 = 0.000000
k1 = 3.000000, k2 = 0.000000
k1 = 4.000000, k2 = 1.000000
```

## See Also

`==`, `>=`, `<=`, `<`, `!=`

## Credits

Example written by Kevin Conder.

## >=

>= — Determines if one value is greater than or equal to another.

## Description

Determines if one value is greater than or equal to another.

## Syntax

```
(a >= b ? v1 : v2)
```

where *a*, *b*, *v1* and *v2* may be expressions, but *a*, *b* not audio-rate.

## Performance

In the above conditional, *a* and *b* are first compared. If the indicated relation is true (*a* greater than or equal to *b*), then the conditional expression has the value of *v1*; if the relation is false, the expression has the value of *v2*.

NB.: If *v1* or *v2* are expressions, these will be evaluated before the conditional is determined.

In terms of binding strength, all conditional operators (i.e. the relational operators (<, etc.), and ?, and :) are weaker than the arithmetic and logical operators (+, -, \*, /, && and //).

These are *operators* not *opcodes*. Therefore, they can be used within orchestra statements, but do not form complete statements themselves.

## Examples

Here is an example of the >= operator. It uses the file *greaterequal.csd* [examples/greaterequal.csd].

### Example 24. Example of the >= operator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o greaterequal.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1.
```

```
instr 1
  ; Get the 4th p-field from the score.
  k1 = p4

  ; Is it greater than or equal to 3? (1 = true, 0 = false)
  k2 = (p4 >= 3 ? 1 : 0)

  ; Print the values of k1 and k2.
  printks "k1 = %f, k2 = %f\\n", 1, k1, k2
endin

</CsInstruments>
<CsScore>

  ; Call Instrument #1 with a p4 = 2.
  i 1 0 0.5 2
  ; Call Instrument #1 with a p4 = 3.
  i 1 1 0.5 3
  ; Call Instrument #1 with a p4 = 4.
  i 1 2 0.5 4
e

</CsScore>
</CsSoundSynthesizer>
```

Its output should include lines like this:

```
k1 = 2.000000, k2 = 0.000000
k1 = 3.000000, k2 = 1.000000
k1 = 4.000000, k2 = 1.000000
```

## See Also

`==, >, <=, <, !=`

## Credits

Example written by Kevin Conder.

## &lt;

< — Determines if one value is less than another.

## Description

Determines if one value is less than another.

## Syntax

```
( a < b ? v1 : v2 )
```

where  $a$ ,  $b$ ,  $v1$  and  $v2$  may be expressions, but  $a$ ,  $b$  not audio-rate.

## Performance

In the above conditional,  $a$  and  $b$  are first compared. If the indicated relation is true ( $a$  less than  $b$ ), then the conditional expression has the value of  $v1$ ; if the relation is false, the expression has the value of  $v2$ .

NB.: If  $v1$  or  $v2$  are expressions, these will be evaluated before the conditional is determined.

In terms of binding strength, all conditional operators (i.e. the relational operators (<, etc.), and ?, and :) are weaker than the arithmetic and logical operators (+, -, \*, /, && and //).

These are *operators* not *opcodes*. Therefore, they can be used within orchestra statements, but do not form complete statements themselves.

## Examples

Here is an example of the < operator. It uses the file *lessthan.csd* [examples/lessthan.csd].

### Example 25. Example of the < operator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o <.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ipch = p4
```

```
ipitch = (ipch < 15 ? cspch(ipch) : ipch) ;if p4 is lower than 15, it assumes p4 to be pitch-class
print ipitch ;and not meant to be a frequency in Hertz
asig poscil .5, ipitch , 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1 ;sine wave

i1 0 3 8.00 ;pitch class
i1 4 3 800 ;frequency

e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
ipitch = 261.626
ipitch = 800.00
```

## See Also

`==, >=, >, <=, !=`

## <=

<= — Determines if one value is less than or equal to another.

## Description

Determines if one value is less than or equal to another.

## Syntax

```
(a <= b ? v1 : v2)
```

where *a*, *b*, *v1* and *v2* may be expressions, but *a*, *b* not audio-rate.

## Performance

In the above conditional, *a* and *b* are first compared. If the indicated relation is true (*a* less than or equal to *b*), then the conditional expression has the value of *v1*; if the relation is false, the expression has the value of *v2*.

NB.: If *v1* or *v2* are expressions, these will be evaluated before the conditional is determined.

In terms of binding strength, all conditional operators (i.e. the relational operators (<, etc.), and ?, and :) are weaker than the arithmetic and logical operators (+, -, \*, /, && and //).

These are *operators* not *opcodes*. Therefore, they can be used within orchestra statements, but do not form complete statements themselves.

## Examples

Here is an example of the <= operator. It uses the file *lessequal.csd* [examples/lessequal.csd].

### Example 26. Example of the <= operator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o <=.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
```

```
kval    randomh 0, 1.2, 20 ;choose between 0 and 1.2

if kval >0 && kval<=.5 then ;3 possible outcomes
  kv1 = 1
elseif kval >.5 && kval<=1 then
  kv1 =2
elseif kval >1 then
  kv1 =3
endif

printks "random number = %f, result = %f\n", .1, kval, kv1
asig    poscil .7, 440*kv1, 1
        outs asig, asig

endin
</CsInstruments>
<CsScore>
f1 0 16384 10 1

i 1 0 5
e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
random number = 1.035781, result = 3.000000
random number = 0.134037, result = 1.000000
random number = 0.130742, result = 1.000000
random number = 1.002550, result = 3.000000
random number = 0.370565, result = 1.000000
random number = 0.655759, result = 2.000000
random number = 0.676154, result = 2.000000
.....
```

## See Also

==, >=, >, <, !=

**\***

\* — Multiplication operator.

## Description

Arithmetic operators perform operations of change-sign (negate), don't-change-sign, logical AND logical OR, add, subtract, multiply and divide. Note that a value or an expression may fall between two of these operators, either of which could take it as its left or right argument, as in

$a + b * c.$

In such cases three rules apply:

1. \* and / bind to their neighbors more strongly than + and -. Thus the above expression is taken as

$a + (b * c)$

with \* taking b and c and then + taking a and b \* c.

2. + and # bind more strongly than &&, which in turn is stronger than ||:

$a \&\& b - c \parallel d$

is taken as

$(a \&\& (b - c)) \parallel d$

3. When both operators bind equally strongly, the operations are done left to right:

$a - b - c$

is taken as

$(a - b) - c$

Parentheses may be used as above to force particular groupings.

## Syntax

`a * b` (no rate restriction)

where the arguments *a* and *b* may be further expressions.

## Arguments

The arguments of `*` can be scalar values or k-rate one dimensional arrays (vectors), or any combination. If one of the arguments is an array, so is the value.

## Examples

Here is an example of the `*` operator. It uses the file `multiplies.csd` [examples/multiplies.csd].

### Example 27. Example of the `*` operator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o multiplies.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kenv      expon 0.01, p3, 1
aout      poscil 0.8*kenv, 440, 1 ;multiply amplitude from 0 to 1 * 0.8
printks "base amplitude * rising expon output = 0.8 * %f\n", .1, kenv
outs     aout, aout
endin

</CsInstruments>
<CsScore>

f 1 0 16384 10 1 ; sine wave

i 1 0 2

e

</CsScore>
</CsoundSynthesizer>
```

Its output should include a line like this:

```
base amplitude * rising expon output = 0.8 * 0.010000
base amplitude * rising expon output = 0.8 * 0.012589
.....
base amplitude * rising expon output = 0.8 * 0.794328
base amplitude * rising expon output = 0.8 * 0.998466
```

## See Also

`-`, `+`, `&&`, `//`, `/`, `^`, `%`

## Credits

Arithmetic on vectors is new in version 6.00

## +

+ — Addition operator

### Description

Arithmetic operators perform operations of change-sign (negate), do-no-change-sign, logical AND logical OR, add, subtract, multiply and divide. Note that a value or an expression may fall between two of these operators, either of which could take it as its left or right argument, as in

$a + b * c$ .

In such cases three rules apply:

1.  $*$  and  $/$  bind to their neighbors more strongly than  $+$  and  $_$ . Thus the above expression is taken as

$a + (b * c)$

with  $*$  taking  $b$  and  $c$  and then  $+$  taking  $a$  and  $b * c$ .

2.  $+$  and  $\#$  bind more strongly than  $\&\&$ , which in turn is stronger than  $\|$ :

$a \&\& b - c \| d$

is taken as

$(a \&\& (b - c)) \| d$

3. When both operators bind equally strongly, the operations are done left to right:

$a - b - c$

is taken as

$(a - b) - c$

Parentheses may be used as above to force particular groupings.

### Syntax

`+a` (no rate restriction)

`a + b` (no rate restriction)

where the arguments  $a$  and  $b$  may be further expressions.

## Arguments

The arguments of + can be scalar values or k-rate one dimensional arrays (vectors), or any combination. If one of the arguments is an array, so is the value.

## Examples

Here is an example of the + operator. It uses the file *adds.csd* [examples/adds.csd].

### Example 28. Example of the + operator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o adds.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
; add unipolar square to oscil
kamp = p4
kcps = 1
itype = 3

klfo lfo kamp, kcps, itype
printk2 klfo
asig oscil 0.7, 440+klfo, 1
outs asig, asig

endin

</CsInstruments>
<CsScore>
;sine wave.
f 1 0 32768 10 1

i 1 0 2 1 ;adds 1 Hz to frequency
i 1 + 2 10 ;adds 10 Hz to frequency
i 1 + 2 220 ;adds 220 Hz to frequency

e

</CsScore>
</CsoundSynthesizer>
```

## See Also

-, &&, //, \*, /, ^, %

## Credits

Arithmetic on vectors is new in version 6.00

-

- — Subtraction operator.

## Description

Arithmetic operators perform operations of change-sign (negate), don't-change-sign, logical AND logical OR, add, subtract, multiply and divide. Note that a value or an expression may fall between two of these operators, either of which could take it as its left or right argument, as in

$a + b * c.$

In such cases three rules apply:

1.  $*$  and  $/$  bind to their neighbors more strongly than  $+$  and  $_$ . Thus the above expression is taken as

$a + (b * c)$

with  $*$  taking  $b$  and  $c$  and then  $+$  taking  $a$  and  $b * c$ .

2.  $+$  and  $#$  bind more strongly than  $\&\&$ , which in turn is stronger than  $\|\|$ :

$a \&\& b - c \|\| d$

is taken as

$(a \&\& (b - c)) \|\| d$

3. When both operators bind equally strongly, the operations are done left to right:

$a - b - c$

is taken as

$(a - b) - c$

Parentheses may be used as above to force particular groupings.

## Syntax

`#a` (no rate restriction)

`a # b` (no rate restriction)

where the arguments *a* and *b* may be further expressions.

## Arguments

The arguments of `_` can be scalar values or k-rate one dimensional arrays (vectors), or any combination. If one of the arguments is an array, so is the value.

## Examples

Here is an example of the `-` operator. It uses the file `subtracts.csd` [examples/subtracts.csd].

### Example 29. Example of the `-` operator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      ;;realtime audio I/O
; For Non-realtime ouput leave only the line below:
; -o -.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

idur = p3
iatt = p4
idec = p5
isus = p3-iatt-idec      ;calculate sustain time from subtracting attack and decay
printf_i "sustain time= note duration - attack - decay --> %.1f-%.1f-%.1f = %.1f\n", 1, idur, iatt, idec

kenv expseg 0.01, iatt, 1, isus, 1, idec, 0.01 ;envelope
asig poscil 1*kenv, 200, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 4096 10 1 ;sine wave
;      attack decay
i 1 0 3 .1 .2
i 1 4 3 .5 1.5
i 1 8 5 4 .5

e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like these:

```
sustain time= note duration - attack - decay --> 3.0-0.1-0.2 = 2.7
```

```
sustain time= note duration - attack - decay --> 3.0-0.5-1.5 = 1.0  
sustain time= note duration - attack - decay --> 5.0-4.0-0.5 = 0.5
```

## See Also

`+`, `&&`, `//`, `*`, `/`, `^`, `%`

## Credits

Arithmetic on vectors is new in version 6.00

# /

/ — Division operator.

## Description

Arithmetic operators perform operations of change-sign (negate), don't-change-sign, logical AND logical OR, add, subtract, multiply and divide. Note that a value or an expression may fall between two of these operators, either of which could take it as its left or right argument, as in

$a + b * c$ .

In such cases three rules apply:

1.  $*$  and  $/$  bind to their neighbors more strongly than  $+$  and  $_$ . Thus the above expression is taken as

$a + (b * c)$

with  $*$  taking  $b$  and  $c$  and then  $+$  taking  $a$  and  $b * c$ .

2.  $+$  and  $\#$  bind more strongly than  $\&\&$ , which in turn is stronger than  $\|$ :

$a \&\& b - c \| d$

is taken as

$(a \&\& (b - c)) \| d$

3. When both operators bind equally strongly, the operations are done left to right:

$a - b - c$

is taken as

$(a - b) - c$

Parentheses may be used as above to force particular groupings.

## Syntax

$a / b$  (no rate restriction)

where the arguments  $a$  and  $b$  may be further expressions.

## Arguments

The arguments of / can be scalar values or k-rate one dimensional arrays (vectors), or any combination. If one of the arguments is an array, so is the value.

## Examples

Here is an example of the / operator. It uses the file *divides.csd* [examples/divides.csd].

### Example 30. Example of the / operator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o ./wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

idiv = 1/p3 * p4
ktrm oscil 1, idiv, 1      ;use oscil as an envelope
printf "retrigger rate per note duration = %f\n",1, idiv
kndx line 5, p3, 1      ;vary index of FM
asig foscil ktrm, 200, 1, 1.4, kndx, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 4096 10 1      ;sine wave

i 1 0 3 10
i 1 4 3 15
i 1 8 3 2

e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like these:

```
retrigger rate per note duration = 3.333333
retrigger rate per note duration = 5.000000
retrigger rate per note duration = 0.666667
```

## See Also

-, +, &&, //, \*, ^, %

## Credits

Arithmetic on vectors is new in version 6.00

**=**

= — Performs a simple assignment.

## Syntax

```
ares = xarg
ires = iarg
kres = karg
ires, ... = iarg, ...
kres, ... = karg, ...
table [ kval ] = karg
```

## Description

Performs a simple assignment.

## Initialization

= (simple assignment) - Put the value of the expression *iarg* (*karg*, *xarg*) into the named result. This provides a means of saving an evaluated result for later use.

From version 5.13 onwards the i- and k-rate versions of assignment can take a number of outputs, and an equal or less number of inputs. If there are less the last value is repeated as necessary.

From version 5.14 values can be assigned to elements of a vector with the square bracket form.

## Examples

Here is an example of the assign opcode. It uses the file *assign.csd* [examples/assign.csd].

### Example 31. Example of the assign opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o assign.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

instr 1
```

```
    ; Assign a value to the variable i1.
    i1 = 1234

    ; Print the value of the i1 variable.
    print i1
  endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
e

</CsScore>
</CsSoundSynthesizer>
```

Its output should include a line like this:

```
instr 1: i1 = 1234.000
```

## See Also

*divz, init, passign, tival*

## Credits

Example written by Kevin Conder.

The extension to multiple values is by

Author: John ffitch  
University of Bath, and Codemist Ltd.  
Bath, UK  
February 2010

New in version 5.13

## **+=**

+= — Performs add and assignment.

## Syntax

```
ares += xarg
ires += iarg
kres += karg
table [ kval ] += karg
```

## Description

Performs an add and assign.

## Initialization

+= - Adds the value of the expression *iarg* (*karg*, *xarg*) into the named result. This provides a means of saving an evaluated result for later use.

## Examples

Here is an example of the plusbecomes opcode. It uses the file *reverb.csd* [examples/reverb.csd].

### Example 32. Example of the plusbecomes opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;;realtime audio out
;-iadc   ;;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o reverb.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 2

ga1 init 0

instr 1
asig poscil .2, cpspch(p4), 1
outs asig, asig

ga1 += asig ;add direct signal to global reverb
```

```
endin

instr 99 ;(highest instr number executed last)

arev reverb gal, 1.5
outs arev, arev

gal = 0 ;clear
endin

</CsInstruments>
<CsScore>
f 1 0 128 10 1 ;sine

i 1 0 0.1 7.00 ;short sounds
i 1 1 0.1 8.02
i 1 2 0.1 8.04
i 1 3 0.1 8.06

i 99 0 6 ;reverb runs for 6 seconds
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*assign*,

## Credits

Author: John ffitch  
University of Bath, and Codemist Ltd.  
Bath, UK  
2013

New in version 6.00

## ==

== — Compares two values for equality.

## Description

Compares two values for equality.

## Syntax

```
(a == b ? v1 : v2)
```

where  $a$ ,  $b$ ,  $v1$  and  $v2$  may be expressions, but  $a$ ,  $b$  not audio-rate.

## Performance

In the above conditional,  $a$  and  $b$  are first compared. If the indicated relation is true ( $a$  is equal to  $b$ ), then the conditional expression has the value of  $v1$ ; if the relation is false, the expression has the value of  $v2$ . (For convenience, a sole "=" will function as "= =".)

NB.: If  $v1$  or  $v2$  are expressions, these will be evaluated before the conditional is determined.

In terms of binding strength, all conditional operators (i.e. the relational operators (<, etc.), and ?, and :) are weaker than the arithmetic and logical operators (+, -, \*, /, && and //).

These are *operators* not *opcodes*. Therefore, they can be used within orchestra statements, but do not form complete statements themselves.

## Examples

Here is an example of the == operator. It uses the file *equals.csd* [examples/equals.csd].

### Example 33. Example of the == operator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o equals.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
```

```
ienv = p4 ;choose envelope in score

if (ienv == 0) kthen
  kenv adsr 0.05, 0.05, 0.95, 0.05 ;sustained envelope
elseif (ienv == 1) kthen
  kenv adsr 0.5, 1, 0.5, 0.5 ;triangular envelope
elseif (ienv == 2) kthen
  kenv adsr 1, 1, 1, 0 ;ramp up
endif

aout vco2 .1, 110, 10
aout = aout * kenv
outs aout, aout

endin

</CsInstruments>
<CsScore>

i1 0 2 0
i1 3 2 1
i1 6 2 2

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

>=, >, <=, <, !=  
s

## ^

^ — “Power of” operator.

## Description

Arithmetic operators perform operations of change-sign (negate), don't-change-sign, logical AND logical OR, add, subtract, multiply and divide. Note that a value or an expression may fall between two of these operators, either of which could take it as its left or right argument, as in

$a + b * c$ .

In such cases three rules apply:

1. \* and / bind to their neighbors more strongly than + and -. Thus the above expression is taken as

$a + (b * c)$

with \* taking b and c and then + taking a and b \* c.

2. + and # bind more strongly than &&, which in turn is stronger than ||:

$a \&\& b - c \parallel d$

is taken as

$(a \&\& (b - c)) \parallel d$

3. When both operators bind equally strongly, the operations are done left to right:

$a - b - c$

is taken as

$(a - b) - c$

Parentheses may be used as above to force particular groupings.

The operator ^ raises *a* to the *b* power. *b* may not be audio-rate. Use with caution as precedence may not work correctly. See *pow*. (New in Csound version 3.493.)

## Syntax

$a \wedge b$  (b not audio-rate)

where the arguments *a* and *b* may be further expressions.

## Examples

Here is an example of the ^ operator. It uses the file *raises.csd* [examples/raises.csd].

### Example 34. Example of the ^ operator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
;-o ^.wav   ; output to audio file
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; Lo-Fi sound

kpow = 10      ;exponent
kbase  line 1, p3, 2.2 ;vary the base
kQuantize = kbase^kpow
kQuantize = kQuantize*0.5 ;half the number of steps for each side of a bipolar signal
printk2 kQuantize
asig  diskin2 "fox.wav", 1, 0, 1 ;loop the fox
asig  = round(asig * kQuantize) / kQuantize ;quantize and scale audio signal
outs  asig, asig

endin
</CsInstruments>
<CsScore>

i1 0 19.2

e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like these:

```
i1      0.50000
i1      0.50021
i1      0.50042
.....
i1 1327.49320
i1 1327.74465
```

## See Also

-, +, &&, //, \*, /, %

# ||

|| — Logical OR operator.

## Description

Arithmetic operators perform operations of change-sign (negate), don't-change-sign, logical AND logical OR, add, subtract, multiply and divide. Note that a value or an expression may fall between two of these operators, either of which could take it as its left or right argument, as in

$a + b * c.$

In such cases three rules apply:

1. \* and / bind to their neighbors more strongly than + and -. Thus the above expression is taken as

$a + (b * c)$

with \* taking b and c and then + taking a and b \* c.

2. + and # bind more strongly than &&, which in turn is stronger than ||:

$a \&\& b - c \parallel d$

is taken as

$(a \&\& (b - c)) \parallel d$

3. When both operators bind equally strongly, the operations are done left to right:

$a - b - c$

is taken as

$(a - b) - c$

Parentheses may be used as above to force particular groupings.

## Syntax

`a || b` (logical OR; not audio-rate)

where the arguments *a* and *b* may be further expressions.

## Examples

Here is an example of the | operator. It uses the file *logicOR.csd* [examples/logicOR.csd].

### Example 35. Example of the | operator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0   ;;realtime audio out and virtual midi keyboard
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
</CsOptions>
<CsInstruments>
;after a UDO from Rory Walsh
sr = 44100
ksmps = 32
nchnls = 2

instr 1 ;displays notes, midi channel and control number information

kstatus, kchan, kdata1, kdata2 midiin
k1 changed kstatus
k2 changed kchan
k3 changed kdata1
k4 changed kdata2
if((k1==1)|| (k2==1)|| (k3==1)|| (k4==1)) then
printks "Value:%d ChanNo:%d CtrlNo:%d\n" , 0, kdata2, kchan, kdata1
endif

endin
</CsInstruments>
<CsScore>

i1 0 60 ;print values for 60 seconds

e
</CsScore>
</CsoundSynthesizer>
```

Its output should include a line like these:

```
Value:127 ChanNo:11 CtrlNo:62
Value:127 ChanNo:11 CtrlNo:60
Value:127 ChanNo:11 CtrlNo:60
Value:127 ChanNo:11 CtrlNo:60
Value:127 ChanNo:11 CtrlNo:60
....
Value:0 ChanNo:11 CtrlNo:62
Value:0 ChanNo:11 CtrlNo:62
Value:0 ChanNo:11 CtrlNo:62
Value:0 ChanNo:11 CtrlNo:62
....
Value:77 ChanNo:11 CtrlNo:23
Value:77 ChanNo:11 CtrlNo:23
Value:76 ChanNo:11 CtrlNo:23
Value:76 ChanNo:11 CtrlNo:23
....
```

## See Also

`-`, `+`, `&&`, `*`, `/`, `^`, `%`

# Odbfs

Odbfs — Sets the value of 0 decibels using full scale amplitude.

## Description

Sets the value of 0 decibels using full scale amplitude.

## Syntax

```
odbfs = iarg
```

```
odbfs
```

## Initialization

*iarg* -- the value of 0 decibels using full scale amplitude.

## Performance

The default is 32767, so all existing orcs *should* work.

Amplitude values in Csound are always relative to a "Odbfs" value representing the peak available amplitude before clipping. In the original Csound, this value was always 32767, corresponding to the bipolar range of a 16bit soundfile or 16bit AD/DA codec. This remains the *default* peak amplitude for Csound, for backward compatibility. The *Odbfs* value enables Csound to produce appropriately scaled values to whatever output format is being used, whether 16bit integer, 24bit integer, 32bit floats, or even 32bit integers.

OdBFS can be defined in the header, to set the amplitude reference Csound will use, but it can also be used as a variable inside instruments like this:

```
ipeak = odbfs
```

```
asig oscil odbfs, freq, 1  
out  asig * 0.3 * odbfs
```

The purpose of the *Odbfs* opcode is for people to start to code Odbfs-relatively (and use the *ampdbfs()* opcodes a lot more!), rather than use explicit sample values. Using Odbfs=1 is in accordance to industry practice, as ranges from -1 to 1 are used in most commercial plugin formats and in most other synthesis systems like Pure Data.

Floats written to a file, when *Odbfs = 1*, will in effect go through no range translation at all. So the numbers in the file are exactly what the orc says they are.

For more details on amplitude values in Csound, see the section *Amplitude values in Csound*

## Example

Here is an example of the Odbfs opcode. It uses the file *Odbfs.csd* [examples/Odbfs.csd].

### Example 36. Example of the Odbfs opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc    -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o 0dbfs.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
ksmps = 10
nchnls = 2

; Set the 0dbfs to 1.
0dbfs = 1

instr 1 ; from linear amplitude (0-1 range)
print p4
a1 oscil p4, 440, 1
outs a1, a1
endin

instr 2 ; from linear amplitude (0-32767 range)
iamp = p4 / 32767
print iamp
a1 oscil iamp, 440, 1
outs a1, a1
endin

instr 3 ; from dB FS
iamp = ampdbfs(p4)
print iamp
a1 oscil iamp, 440, 1
outs a1, a1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

i 1 0 1 1
i 1 + 1 0.5
i 1 + 1 0.1
s
i 2 0 1 32767
i 2 + 1 [32767/2]
i 2 + 1 [3276.7]
s
i 3 0 1 0
i 3 + 1 -6
i 3 + 1 -20
e

</CsScore>
</CsoundSynthesizer>

```

## See also

*ampdbfs()*

## Credits

Author: Richard Dobson  
May 2002

New in version 4.10

# A4

A4 — Sets the base frequency for pitch A4.

## Description

These statements are global value *assignments*, made at the beginning of an orchestra, before any instrument block is defined. Their function is to set certain *reserved symbol variables* that are required for performance. Once set, these reserved symbols can be used in expressions anywhere in the orchestra.

## Syntax

```
A4 = iarg
```

## Initialization

*A4* = (optional) -- set reference frequency for pitch A4 to *iarg* Hertz. The default value is 440.

In addition, any *global variable* can be initialized by an *init-time assignment* anywhere before the first *instr statement*. All of the above assignments are run as instrument 0 (i-pass only) at the start of real performance.

Beginning with Csound version 6.08, *A4* may be used. It affects the behaviour of the opcodes *cpspch*, *cpsoct*, *cps2pch*, *cpsxpch* and *cpsmidinn*.

## See Also

*sr*, *kr*, *ksmps*, *nchnls*, *nchnls\_i*, *0dbfs*, *cpspch*, *cpsoct*, *cpsmidinn*, *cps2pch*, *cpsxpch*.

## Credits

Authors: John fitch  
September 2016

New in Csound Version 6.08

## &lt;&lt;

<< — Bitshift left operator.

## Description

The bitshift operators shift the bits to the left or to the right the number of bits given.

The priority of these operators is less binding than the arithmetic ones, but more binding than the comparisons.

Parentheses may be used as above to force particular groupings.

## Syntax

```
a << b (bitshift left)
```

where the arguments *a* and *b* may be further expressions.

## Examples

Here is an example of the bitshift left operator. It uses the file *bitshift.csd* [examples/bitshift.csd].

### Example 37. Example of the bitshift left operator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
;-odac      -iadc      ;;RT audio I/O
; For Non-realtime output leave only the line below:
-o bitshift.wav -W --nosound ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 128
nchnls = 2

instr 1 ;bit shift right
ival = p4>>p5
printf_i "%i>>%i = %i\n", 1, p4, p5, ival
endin

instr 2 ;bit shift left
ival = p4<<p5
printf_i "%i<<%i = %i\n", 1, p4, p5, ival
endin

</CsInstruments>
<CsScore>
i 1 0 0.1 2 1
i 1 + . 3 1
i 1 + . 7 2
```

```

i 1 + . 16 1
i 1 + . 16 2
i 1 + . 16 3

i 2 5 0.1 1 1
i 2 + . 1 2
i 2 + . 1 3
i 2 + . 1 4
i 2 + . 2 1
i 2 + . 2 2
i 2 + . 2 3
i 2 + . 3 2
e
</CsScore>
</CsoundSynthesizer>

```

The example above will produce the following output:

```

2>>1 = 1
B 0.000 .. 0.100 T 0.100 TT 0.100 M: 0.0 0.0
3>>1 = 1
B 0.100 .. 0.200 T 0.200 TT 0.200 M: 0.0 0.0
7>>2 = 1
B 0.200 .. 0.300 T 0.300 TT 0.300 M: 0.0 0.0
16>>1 = 8
B 0.300 .. 0.400 T 0.400 TT 0.400 M: 0.0 0.0
16>>2 = 4
B 0.400 .. 0.500 T 0.500 TT 0.500 M: 0.0 0.0
16>>3 = 2
B 0.500 .. 5.000 T 5.000 TT 5.000 M: 0.0 0.0
new alloc for instr 2:
1<<1 = 2
B 5.000 .. 5.100 T 5.100 TT 5.100 M: 0.0 0.0
1<<2 = 4
B 5.100 .. 5.200 T 5.200 TT 5.200 M: 0.0 0.0
1<<3 = 8
B 5.200 .. 5.300 T 5.300 TT 5.300 M: 0.0 0.0
1<<4 = 16
B 5.300 .. 5.400 T 5.400 TT 5.400 M: 0.0 0.0
2<<1 = 4
B 5.400 .. 5.500 T 5.500 TT 5.500 M: 0.0 0.0
2<<2 = 8
B 5.500 .. 5.600 T 5.600 TT 5.600 M: 0.0 0.0
2<<3 = 16
B 5.600 .. 5.700 T 5.700 TT 5.700 M: 0.0 0.0
3<<2 = 12

```

## See Also

>>, &, /#

## >>

>> — Bitshift right operator.

### Description

The bitshift operators shift the bits to the left or to the right the number of bits given.

The priority of these operators is less binding than the arithmetic ones, but more binding than the comparisons.

Parentheses may be used as above to force particular groupings.

### Syntax

```
a >> b (bitshift left)
```

where the arguments *a* and *b* may be further expressions.

### Examples

See the entry for the << operator for an example.

### See Also

<<, &, /#

# &

& — Bitwise AND operator.

## Description

The bitwise operators perform operations of bitwise AND, bitwise OR, bitwise NOT and bitwise non-equivalence.

## Syntax

```
a & b (bitwise AND)
```

where the arguments *a* and *b* may be further expressions. They are converted to the nearest integer to machine precision and then the operation is performed.

## Performance

The priority of these operators is less binding than the arithmetic ones, but more binding than the comparisons.

Parentheses may be used as above to force particular groupings.

## Examples

Here is an example of the bitwise AND and OR operators. It uses the file *bitwise.csd* [examples/bitwise.csd].

### Example 38. Example of the bitwise operators.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

instr 1
ireultOr = p4 | p5
ireultAnd = p4 & p5
prints "%i | %i = %i\\n", p4, p5, ireultOr
prints "%i & %i = %i\\n", p4, p5, ireultAnd
endin

instr 2 ; decimal to binary converter
Sbinary = ""
inumbits = 8
icount init inumbits - 1
```

```
pass:

  ivalue = 2 ^ icount
  if (p4 & ivalue >= ivalue) then
    Sdigit = "1"
  else
    Sdigit = "0"
  endif
  Sbinary strcat Sbinary, Sdigit

loop_ge icount, 1, 0, pass

Stext sprintf "%i is %s in binary\\n", p4, Sbinary
prints Stext
endin

</CsInstruments>
<CsScore>
i 1 0 0.1 1 2
i 1 + . 1 3
i 1 + . 2 4
i 1 + . 3 10

i 2 2 0.1 12
i 2 + . 9
i 2 + . 15
i 2 + . 49

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

|, #, ▯

# |

| — Bitwise OR operator.

## Description

The bitwise operators perform operations of bitwise AND, bitwise OR, bitwise NOT and bitwise non-equivalence.

## Syntax

`a | b` (bitwise OR)

where the arguments *a* and *b* may be further expressions. They are converted to the nearest integer to machine precision and then the operation is performed.

## Performance

The priority of these operators is less binding than the arithmetic ones, but more binding than the comparisons.

Parentheses may be used as above to force particular groupings.

For an example of usage, see the entry for `&`

## See Also

`&`, `#`, `¬`

**¬**

¬ — Bitwise NOT operator.

## Description

The bitwise operators perform operations of bitwise AND, bitwise OR, bitwise NOT and bitwise non-equivalence.

The priority of these operators is less binding than the arithmetic ones, but more binding than the comparisons.

Parentheses may be used as above to force particular groupings.

## Syntax

`~ a` (bitwise NOT)

where the argument *a* may be a further expression. It is converted to the nearest integer to machine precision and then the operation is performed.

## See Also

`&, / #`

## #

# — Bitwise NON EQUIVALENCE operator.

### Description

The bitwise operators perform operations of bitwise AND, bitwise OR, bitwise NOT and bitwise non-equivalence.

The priority of these operators is less binding than the arithmetic ones, but more binding than the comparisons.

Parentheses may be used as above to force particular groupings.

### Syntax

```
a # b (bitwise NON EQUIVALENCE)
```

where the arguments *a* and *b* may be further expressions. They are converted to the nearest integer to machine precision and then the operation is performed.

### See Also

&, /, ¬

## a

a — Converts a k-rate parameter to an a-rate value with interpolation.

## Description

Converts a k-rate parameter to an a-rate value with interpolation.

## Syntax

`a(x)` (control-rate args only)

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

## Examples

Here is an example of the a opcode. It uses the file `opa.csd` [examples/opa.csd].

### Example 39. Example of the a opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
;-o a.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; hear the difference between instr.1 and 2
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;sine wave at k-rate
ksig oscil 0.8, 440, 1
; k-rate to the audio-rate conversion
asig = a(ksig)
outs asig, asig

endin

instr 2 ;sine wave at a-rate
asig oscil 0.8, 440, 1
outs asig, asig

endin

</CsInstruments>
```

```
<CsScore>
; sine wave.
f 1 0 16384 10 1

i 1 0 2
i 2 2 2
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*i, k*

## Credits

Author: Gabriel Maldonado

New in version 4.21

# abs

abs — Returns an absolute value.

## Description

Returns the absolute value of  $x$ .

## Syntax

`abs(x)` (no rate restriction)

`abs(k/i[])` (k- or i-arrays )

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

## Examples

Here is an example of the abs opcode. It uses the file `abs.csd` [examples/abs.csd].

### Example 40. Example of the abs opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsoundOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o abs.wav -W ;; for file output any platform
</CsoundOptions>
<CsoundInstruments>

sr = 44100
ksmps = 32
nchnls = 2

instr 1
ix = p4
iabs = abs(ix)
print iabs

endin

</CsoundInstruments>
<CsoundScore>

i 1 0 1 0
i 1 + 1 -.15
i 1 + 1 -13
e

</CsoundScore>
```

```
</CsoundSynthesizer>
```

Its output should include lines like:

```
instr 1: iabs = 0.000  
instr 1: iabs = 0.150  
instr 1: iabs = 13.000
```

## See Also

*exp, frac, int, log, log10, i, sqrt*

## active

`active` — Returns the number of active instances of an instrument.

## Description

Returns the number of active instances of an instrument with options to ignore releasing instances.

## Syntax

```
ir active insnum [,iopt [,inorel]]  
ir active Sinsname [,iopt [,inorel]]  
kres active kinsnum [,iopt [,inorel]]
```

## Initialization

*insnum* -- number or string name of the instrument to be reported

*Sinsname* -- instrument name

*iopt* -- select currently active (zero, default), or all every active (non zero)

*inorel* -- if non-zero ignore instruments in release phase (zero, default), only valid if *iopts* is zero.

## Performance

*kinsnum* -- number or string name of the instrument to be reported

*active* returns the number of active instances of instrument number *insnum/kinsnum* (or named instrument *Sinsname*). As of Csound 4.17 the output is updated at k-rate (if input arg is k-rate), to allow running count of instr instances.

As of Csound 5.17 if the instrument number is given as zero then all instruments are counted.

## Examples

Here is a simple example of the active opcode. It uses the file *active.csd* [examples/active.csd].

### Example 41. Simple example of the active opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
; Audio out  Audio in  
-odac          -iadc      ;;RT audio I/O  
; For Non-realtime ouput leave only the line below:  
; -o active.wav -W ;; for file output any platform  
</CsOptions>
```

```

<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - a noisy waveform.
instr 1
; Generate a really noisy waveform.
anoisy rand 44100
; Turn down its amplitude.
aoutput gain anoisy, 2500
; Send it to the output.
out aoutput
endin

; Instrument #2 - counts active instruments.
instr 2
; Count the active instances of Instrument #1.
icount active 1
; Print the number of active instances.
print icount
endin

</CsInstruments>
<CsScore>

; Start the first instance of Instrument #1 at 0:00 seconds.
i 1 0.0 3.0

; Start the second instance of Instrument #1 at 0:015 seconds.
i 1 1.5 1.5

; Play Instrument #2 at 0:01 seconds, when we have only
; one active instance of Instrument #1.
i 2 1.0 0.1

; Play Instrument #2 at 0:02 seconds, when we have
; two active instances of Instrument #1.
i 2 2.0 0.1
e

</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like this:

```

instr 2: icount = 1.000
instr 2: icount = 2.000

```

Here is a more advanced example of the active opcode. It displays the results of the active opcode at k-rate instead of i-rate. It uses the file *active\_k.csd* [examples/active\_k.csd].

### Example 42. Example of the active opcode at k-rate.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform

```

```

; Audio out   Audio in
-odac         -iadc   ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o active_k.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - a noisy waveform.
instr 1
  ; Generate a really noisy waveform.
  anoisy rand 44100
  ; Turn down its amplitude.
  aoutput gain anoisy, 2500
  ; Send it to the output.
  out aoutput
endin

; Instrument #2 - counts active instruments at k-rate.
instr 2
  ; Count the active instances of Instrument #1.
  kcount active 1
  ; Print the number of active instances.
  printk2 kcount
endin

</CsInstruments>
<CsScore>

; Start the first instance of Instrument #1 at 0:00 seconds.
i 1 0.0 3.0

; Start the second instance of Instrument #1 at 0:015 seconds.
i 1 1.5 1.5

; Play Instrument #2 at 0:01 seconds, when we have only
; one active instance of Instrument #1.
i 2 1.0 0.1

; Play Instrument #2 at 0:02 seconds, when we have
; two active instances of Instrument #1.
i 2 2.0 0.1
e

</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like:

```

i2      1.00000
i2      2.00000

```

Here is another example of the active opcode, using the number of instances to calculate gain. It uses the file *active\_scale.csd* [examples/active\_scale.csd].

### Example 43. Example of the active opcode at k-rate.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o atone.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr= 44100
ksmps = 64
nchnls = 1
0dbfs = 1

;by Victor Lazzarini 2008

instr 1
kscal active 1
kamp port 1/kscal, 0.01
asig oscili kamp, p4, 1
kenv linseg 0, 0.1,1,p3-0.2,1,0.1, 0

        out asig*kenv
endin

</CsInstruments>
<CsScore>
f1 0 16384 10 1

i1 0 10 440
i1 1 3 220
i1 2 5 350
i1 4 3 700
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitc  
University of Bath/Codemist Ltd.  
Bath, UK  
July, 1999

Examples written by Kevin Conder.

New in Csound version 3.57; named instruments added version 5.13

Option for all ever active new in 5.13

Count of all instruments new in 5.17

No release option new in 5.19

## adsr

`adsr` — Calculates the classical ADSR envelope using linear segments.

## Description

Calculates the classical ADSR envelope using linear segments.

## Syntax

```
ares adsr iatt, idec, islev, irel [, idel]
```

```
kres adsr iatt, idec, islev, irel [, idel]
```

## Initialization

*iatt* -- duration of attack phase

*idec* -- duration of decay

*islev* -- level for sustain phase

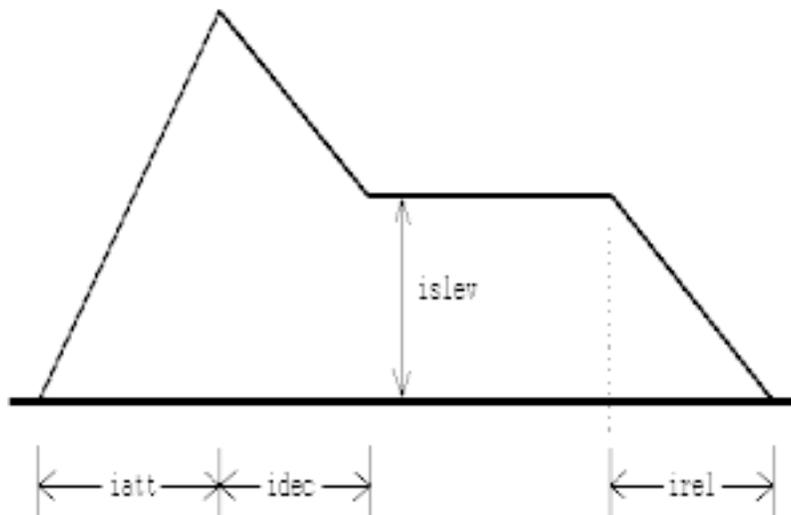
*irel* -- duration of release phase

*idel* -- period of zero before the envelope starts

## Performance

The envelope generated is the range 0 to 1 and may need to be scaled further, depending on the amplitude required. If using *Odbfs* = 1, scaling down will probably be required since playing more than one note might result in clipping. If not using *Odbfs*, scaling to a large amplitude (e.g. 32000) might be required.

The envelope may be described as:



Picture of an ADSR envelope.

The length of the sustain is calculated from the length of the note. This means *adsr* is not suitable for use with MIDI events. The opcode *madsr* uses the *linsegr* mechanism, and so can be used in MIDI applications.

*adsr* is new in Csound version 3.49.

## Examples

Here is an example of the *adsr* opcode. It uses the file *adsr.csd* [examples/adsr.csd].

### Example 44. Example of the *adsr* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o adsr.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

iatt = p5
idec = p6
islev = p7
irel = p8

kenv adsr iatt, idec, islev, irel
kcps = cpspch(p4) ;frequency

asig vco2 kenv * 0.8, kcps
outs asig, asig

endin

</CsInstruments>
<CsScore>

i 1 0 2 7.00 .0001 1 .5 .001 ; short attack
i 1 3 2 7.02 1 .5 .5 .001 ; long attack
i 1 6 2 6.09 .0001 1 .5 .7 ; long release

e

</CsScore>
</CsoundSynthesizer>

```

## See Also

*madsr*, *mxadsr*, *xadsr*

## Credits

Author: John ffitch

New in version 3.49

# adsyn

adsyn — Output is an additive set of individually controlled sinusoids, using an oscillator bank.

## Description

Output is an additive set of individually controlled sinusoids, using an oscillator bank.

## Syntax

```
ares adsyn kamod, kfmmod, ksmmod, ifilcod
```

## Initialization

*ifilcod* -- integer or character-string denoting a control-file derived from analysis of an audio signal. An integer denotes the suffix of a file *adsyn.m* or *pvoc.m*; a character-string (in double quotes) gives a filename, optionally a full pathname. If not fullpath, the file is sought first in the current directory, then in the one given by the environment variable *SADIR* (if defined). *adsyn* control contains breakpoint amplitude- and frequency-envelope values organized for oscillator resynthesis, while *pvoc* control contains similar data organized for fft resynthesis. Memory usage depends on the size of the files involved, which are read and held entirely in memory during computation but are shared by multiple calls (see also *lpread*).

## Performance

*kamod* -- amplitude factor of the contributing partials.

*kfmmod* -- frequency factor of the contributing partials. It is a control-rate transposition factor: a value of 1 incurs no transposition, 1.5 transposes up a perfect fifth, and .5 down an octave.

*ksmmod* -- speed factor of the contributing partials.

*adsyn* synthesizes complex time-varying timbres through the method of additive synthesis. Any number of sinusoids, each individually controlled in frequency and amplitude, can be summed by high-speed arithmetic to produce a high-fidelity result.

Component sinusoids are described by a control file describing amplitude and frequency tracks in millisecond breakpoint fashion. Tracks are defined by sequences of 16-bit binary integers:

```
-1, time, amp, time, amp,...  
-2, time, freq, time, freq,...
```

such as from hetrodyne filter analysis of an audio file. (For details see *hetro*.) The instantaneous amplitude and frequency values are used by an internal fixed-point oscillator that adds each active partial into an accumulated output signal. While there is a practical limit (limit removed in version 3.47) on the number of contributing partials, there is no restriction on their behavior over time. Any sound that can be described in terms of the behavior of sinusoids can be synthesized by *adsyn* alone.

Sound described by an *adsyn* control file can also be modified during re-synthesis. The signals *kamod*, *kfmmod*, *ksmmod* will modify the amplitude, frequency, and speed of contributing partials. These are multiplying factors, with *kfmmod* modifying the frequency and *ksmmod* modifying the *speed* with which the mil-

lisecond breakpoint line-segments are traversed. Thus .7, 1.5, and 2 will give rise to a softer sound, a perfect fifth higher, but only half as long. The values 1,1,1 will leave the sound unmodified. Each of these inputs can be a control signal.

## Examples

Here is an example of the `adsyn` opcode. It uses the file `adsyn.csd` [examples/adsyn.csd], and `kickroll.het` [examples/kickroll.het]. The file “kickroll.het” was created by using the `hetro` utility with the audio file `kickroll.wav` [examples/kickroll.wav].

### Example 45. Example of the `adsyn` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o adsyn.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
; analyze the file "kickroll.wav" first
kamod = 1
kfmod = p4
ksmod = p5

asig adsyn kamod, kfmod, ksmod, "kickroll.het"
outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 4 1 .2
i 1 + 1 2 1
i 1 + 1 .3 1.5
e

</CsScore>
</CsoundSynthesizer>
```

# adsynt

adsynt — Performs additive synthesis with an arbitrary number of partials, not necessarily harmonic.

## Description

Performs additive synthesis with an arbitrary number of partials, not necessarily harmonic.

## Syntax

```
ares adsynt kamp, kcps, iwfn, ifreqfn, iampfn, icnt [, iphs]
```

## Initialization

*iwfn* -- table containing a waveform, usually a sine. Table values are not interpolated for performance reasons, so larger tables provide better quality.

*ifreqfn* -- table containing frequency values for each partial. *ifreqfn* may contain beginning frequency values for each partial, but is usually used for generating parameters at runtime with *tablew*. Frequencies must be relative to *kcps*. Size must be at least *icnt*.

*iampfn* -- table containing amplitude values for each partial. *iampfn* may contain beginning amplitude values for each partial, but is usually used for generating parameters at runtime with *tablew*. Amplitudes must be relative to *kamp*. Size must be at least *icnt*.

*icnt* -- number of partials to be generated

*iphs* -- initial phase of each oscillator, if *iphs* = -1, initialization is skipped. If *iphs* > 1, all phases will be initialized with a random value.

## Performance

*kamp* -- amplitude of note

*kcps* -- base frequency of note. Partial frequencies will be relative to *kcps*.

Frequency and amplitude of each partial is given in the two tables provided. The purpose of this opcode is to have an instrument generate synthesis parameters at k-rate and write them to global parameter tables with the *tablew* opcode.

## Examples

Here is an example of the adsynt opcode. It uses the file *adsynt.csd* [examples/adsynt.csd]. These two instruments perform additive synthesis. The output of each sounds like a Tibetan bowl. The first one is static, as parameters are only generated at init-time. In the second one, parameters are continuously changed.

### Example 46. Example of the adsynt opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o adsynt.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
Odbfs = 1
; Generate a sinewave table.
giwave ftgen 1, 0, 1024, 10, 1
; Generate two empty tables for adsynt.
gifrqs ftgen 2, 0, 32, 7, 0, 32, 0
; A table for frequency and amp parameters.
giamps ftgen 3, 0, 32, 7, 0, 32, 0

; Generates parameters at init time
instr 1
; Generate 10 voices.
icnt = 10
; Init loop index.
index = 0

; Loop only executed at init time.
loop:
; Define non-harmonic partials.
ifreq pow index + 1, 1.5
; Define amplitudes.
iamp = 1 / (index+1)
; Write to tables.
tableiw ifreq, index, gifrqs
; Used by adsynt.
tableiw iamp, index, giamps

index = index + 1
; Do loop/
if (index < icnt) igoto loop

asig adsynt 0.3, 150, giwave, gifrqs, giamps, icnt
outs asig, asig
endin

; Generates parameters every k-cycle.
instr 2
; Generate 10 voices.
icnt = 10
; Reset loop index.
kindex = 0

; Loop executed every k-cycle.
loop:
; Generate lfo for frequencies.
kspeed pow kindex + 1, 1.6
; Individual phase for each voice.
kphas phasorbk kspeed * 0.7, kindex, icnt
klfo table kphas, giwave, 1
; Arbitrary parameter twiddling...
kdepth pow 1.4, kindex
kfreq pow kindex + 1, 1.5
kfreq = kfreq + klfo*0.006*kdepth

```

```
; Write freqs to table for adsynt.
tablew kfreq, kindex, gifrqs

; Generate lfo for amplitudes.
ksped pow kindex + 1, 0.8
; Individual phase for each voice.
kphas phasorbk ksped*0.13, kindex, icnt, 2
klfo table kphas, giwave, 1
; Arbitrary parameter twiddling...
kamp pow 1 / (kindex + 1), 0.4
kamp = kamp * (0.3+0.35*(klfo+1))

; Write amps to table for adsynt.
tablew kamp, kindex, giamps

kindex = kindex + 1
; Do loop.
if (kindex < icnt) kgoto loop

asig adsynt 0.25, 150, giwave, gifrqs, giamps, icnt
outs asig, asig
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for 2.5 seconds.
i 1 0 2.5
; Play Instrument #2 for 2.5 seconds.
i 2 3 2.5
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Peter Neubäcker  
Munich, Germany  
August, 1999

New in Csound version 3.58

# adsynt2

adsynt2 — Performs additive synthesis with an arbitrary number of partials -not necessarily harmonic- with interpolation.

## Description

Performs additive synthesis with an arbitrary number of partials, not necessarily harmonic. (see *adsynt* for detailed manual)

## Syntax

```
ar adsynt2 kamp, kcps, iwfn, ifreqfn, iampfn, icnt [, iphs]
```

## Initialization

*iwfn* -- table containing a waveform, usually a sine. Table values are not interpolated for performance reasons, so larger tables provide better quality.

*ifreqfn* -- table containing frequency values for each partial. *ifreqfn* may contain beginning frequency values for each partial, but is usually used for generating parameters at runtime with *tablew*. Frequencies must be relative to *kcps*. Size must be at least *icnt*.

*iampfn* -- table containing amplitude values for each partial. *iampfn* may contain beginning amplitude values for each partial, but is usually used for generating parameters at runtime with *tablew*. Amplitudes must be relative to *kamp*. Size must be at least *icnt*.

*icnt* -- number of partials to be generated

*iphs* -- initial phase of each oscillator, if *iphs* = -1, initialization is skipped. If *iphs* > 1, all phases will be initialized with a random value.

## Performance

*kamp* -- amplitude of note

*kcps* -- base frequency of note. Partial frequencies will be relative to *kcps*.

Frequency and amplitude of each partial is given in the two tables provided. The purpose of this opcode is to have an instrument generate synthesis parameters at k-rate and write them to global parameter tables with the *tablew* opcode.

*adsynt2* is identical to *adsynt* (by Peter Neubäcker), except it provides linear interpolation for amplitude envelopes of each partial. It is a bit slower than *adsynt*, but interpolation highly improves sound quality in fast amplitude envelope transients when  $kr < sr$  (i.e. when  $ksmps > 1$ ). No interpolation is provided for pitch envelopes, since in this case sound quality degradation is not so evident even with high values of *ksmps*. It is not recommended when  $kr = sr$ , in this case *adsynt* is better (since it is faster).

## Examples

Here is an example of the *adsynt2* opcode. It uses the file *adsynt2.csd* [examples/adsynt2.csd]. These two instruments perform additive synthesis. The output of each sounds like a Tibetan bowl. The first one

is static, as parameters are only generated at init-time. In the second one, parameters are continuously changed.

### Example 47. Example of the adsynt2 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o adsynt2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
; Generate a sinewave table.
giwave ftgen 1, 0, 1024, 10, 1
; Generate two empty tables for adsynt2.
gifrqs ftgen 2, 0, 32, 7, 0, 32, 0
; A table for frequency and amp parameters.
giamps ftgen 3, 0, 32, 7, 0, 32, 0

; Generates parameters at init time
instr 1
; Generate 10 voices.
icnt = 10
; Init loop index.
index = 0

; Loop only executed at init time.
loop:
; Define non-harmonic partials.
ifreq pow index + 1, 1.5
; Define amplitudes.
iamp = 1 / (index+1)
; Write to tables.
tableiw ifreq, index, gifrqs
; Used by adsynt2.
tableiw iamp, index, giamps

index = index + 1
; Do loop/
if (index < icnt) igoto loop

asig adsynt2 0.4, 150, giwave, gifrqs, giamps, icnt
outs asig, asig
endin

; Generates parameters every k-cycle.
instr 2
; Generate 10 voices.
icnt = 10
; Reset loop index.
kindex = 0

; Loop executed every k-cycle.
loop:
; Generate lfo for frequencies.

```

```

kspeed pow kindex + 1, 1.6
; Individual phase for each voice.
kphas phasorbk kspeed * 0.7, kindex, icnt
klfo table kphas, giwave, 1
; Arbitrary parameter twiddling...
kdepth pow 1.4, kindex
kfreq pow kindex + 1, 1.5
kfreq = kfreq + klfo*0.006*kdepth

; Write freqs to table for adsynt2.
tablew kfreq, kindex, gifrqs

; Generate lfo for amplitudes.
kspeed pow kindex + 1, 0.8
; Individual phase for each voice.
kphas phasorbk kspeed*0.13, kindex, icnt, 2
klfo table kphas, giwave, 1
; Arbitrary parameter twiddling...
kamp pow 1 / (kindex + 1), 0.4
kamp = kamp * (0.3+0.35*(klfo+1))

; Write amps to table for adsynt2.
tablew kamp, kindex, giamps

kindex = kindex + 1
; Do loop.
if (kindex < icnt) kgoto loop

asig adsynt2 0.25, 150, giwave, gifrqs, giamps, icnt
outs asig, asig
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for 2.5 seconds.
i 1 0 2.5
; Play Instrument #2 for 2.5 seconds.
i 2 3 2.5
e

</CsScore>
</CsoundSynthesizer>

```

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# aftouch

aftouch — Get the current after-touch value for this channel.

## Description

Get the current after-touch value for this channel.

## Syntax

```
kaft aftouch [imin] [, imax]
```

## Initialization

*imin* (optional, default=0) -- minimum limit on values obtained.

*imax* (optional, default=127) -- maximum limit on values obtained.

## Performance

Get the current after-touch value for this channel.

## Examples

Here is an example of the aftouch opcode. It uses the file *aftouch.csd* [examples/aftouch.csd].

### Example 48. Example of the aftouch opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  No messages  MIDI in
-odac          -d          -M0   ;;RT audio out with MIDI in
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kaft aftouch 0, 1
printk2 kaft

;aftertouch from music keyboard used for volume control
asig oscil 0.7 * kaft, 220, 1
outs asig, asig

endin
```

```
</CsInstruments>
<CsScore>
; sine wave.
f 1 0 16384 10 1

i 1 0 30
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*ampmidi, cpsmidi, cpsmidib, midictrl, notnum, octmidi, octmidib, pchbend, pchmidi, pchmidib, veloc*

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

# alpass

alpass — Reverberates an input signal with a flat frequency response.

## Description

Reverberates an input signal with a flat frequency response.

## Syntax

```
ares alpass asig, xrvt, ilpt [, iskip] [, insmps]
```

## Initialization

*ilpt* -- loop time in seconds, which determines the “echo density” of the reverberation. This in turn characterizes the “color” of the filter whose frequency response curve will contain  $ilpt * sr/2$  peaks spaced evenly between 0 and  $sr/2$  (the Nyquist frequency). Loop time can be as large as available memory will permit. The space required for an  $n$  second loop is  $4n*sr$  bytes. The delay space is allocated and returned as in *delay*.

*iskip* (optional, default=0) -- initial disposition of delay-loop data space (cf. *reson*). The default value is 0.

*insmps* (optional, default=0) -- delay amount, as a number of samples.

## Performance

*xrvt* -- the reverberation time (defined as the time in seconds for a signal to decay to 1/1000, or 60dB down from its original amplitude).

This filter reiterates the input with an echo density determined by loop time *ilpt*. The attenuation rate is independent and is determined by *xrvt*, the reverberation time (defined as the time in seconds for a signal to decay to 1/1000, or 60dB down from its original amplitude). Output will begin to appear immediately.

## Examples

Here is an example of the alpass opcode. It uses the file *alpass.csd* [examples/alpass.csd].

### Example 49. Example of the alpass opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o alpass.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```
sr = 44100
ksmps = 100
nchnls = 2
0dbfs = 1

gamix init 0

instr 1

acps   expon p4, p3, p5
asig vco 0.6, acps, 1
outs  asig, asig

gamix = gamix + asig

endin

instr 99

arvt1 line 3.5*1.5, p3, 6
arvt2 line 3.5, p3, 4
ilpt = 0.1
aleft alpass gamix, arvt1, ilpt
aright alpass gamix, arvt2, ilpt*2
outs  aleft, aright

gamix = 0 ; clear mixer

endin

</CsInstruments>
<CsScore>
f1 0 4096 10 1
i 1 0 3 20 2000

i 99 0 8
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*comb, reverb, valpass, vcomb*

## Credits

Audio rate parameter introduced in version 6.02

November 2013.

# alwayson

`alwayson` — Activates the indicated instrument in the orchestra header, without need for an `i` statement.

## Description

Activates the indicated instrument in the orchestra header, without need for an `i` statement. Instruments must be activated in the same order as they are defined.

The `alwayson` opcode is designed to simplify the definition of re-usable orchestras with signal processing or effects chains and networks.

## Syntax

```
alwayson Tinstrument [p4, ..., pn]
```

## Initialization

*Tinstrument* -- String name of the instrument definition to be turned on.

*[p4, ..., pn]* -- Optional pfields to be passed to the instrument, in the same order and type as if this were an `i` statement.

When the instrument is activated, `p1` is the `insno`, `p2` is 0, and `p3` is -1. Pfields from `p4` on may optionally be sent to the instrument.

## Examples

Here is an example of the `alwayson` opcode. It uses the file `alwayson.csd` [examples/alwayson.csd].

### Example 50. Example of the `alwayson` opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
;-Wfo alwayson.wav
</CsOptions>
<CsInstruments>

; Initialize the global variables.

sr      = 44100
ksmps  = 32
nchnls = 2

; Connect up instruments and effects to create the signal flow graph.

connect "SimpleSine",    "leftout",    "Reverberator",    "leftin"
connect "SimpleSine",    "rightout",   "Reverberator",    "rightin"

connect "Moogy",        "leftout",    "Reverberator",    "leftin"
connect "Moogy",        "rightout",   "Reverberator",    "rightin"

connect "Reverberator",  "leftout",    "Compressor",      "leftin"
```

```

connect "Reverberator", "rightout", "Compressor", "rightin"

connect "Compressor", "leftout", "Soundfile", "leftin"
connect "Compressor", "rightout", "Soundfile", "rightin"

; Turn on the "effect" units in the signal flow graph.

alwayson "Reverberator", 0.91, 12000
alwayson "Compressor"
alwayson "Soundfile"

; Define instruments and effects in order of signal flow.

instr SimpleSine
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ; Default values: p1 p2 p3 p4 p5 p6 p7 p8 p9 p10
    pset 0, 0, 10, 0, 0, 0, 0.5
    iattack = 0.015
    idecay = 0.07
    isustain = p3
    irelease = 0.3
    p3 = iattack + idecay + isustain + irelease
    adamping linsegr 0.0, iattack, 1.0, idecay + isustain, 1.0, irelease, 0.0
    iHz = cpsmidinn(p4)
    ; Rescale MIDI velocity range to a musically usable range of dB.
    iamplitude = ampdb(p5 / 127 * 15.0 + 60.0)
    ; Use ftgenonce instead of ftgen, ftgentmp, or f statement.
    icosine ftgenonce 0, 0, 65537, 11, 1
    aoscili oscili iamplitude, iHz, icosine
    aadsr madsr iattack, idecay, 0.6, irelease
    asignal = aoscili * aadsr
    aleft, aright pan2 asignal, p7
    ; Stereo audio output to be routed in the orchestra header.
    outleta "leftout", aleft
    outleta "rightout", aright
    endin

instr Moogy
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ; Default values: p1 p2 p3 p4 p5 p6 p7 p8 p9 p10
    pset 0, 0, 10, 0, 0, 0, 0.5
    iattack = 0.003
    isustain = p3
    irelease = 0.05
    p3 = iattack + isustain + irelease
    adamping linsegr 0.0, iattack, 1.0, isustain, 1.0, irelease, 0.0
    iHz = cpsmidinn(p4)
    ; Rescale MIDI velocity range to a musically usable range of dB.
    iamplitude = ampdb(p5 / 127 * 20.0 + 60.0)
    print iHz, iamplitude
    ; Use ftgenonce instead of ftgen, ftgentmp, or f statement.
    isine ftgenonce 0, 0, 65537, 10, 1
    asignal vco iamplitude, iHz, 1, 0.5, isine
    kfco line 2000, p3, 200
    krez = 0.8
    asignal moogvcf asignal, kfco, krez, 100000
    asignal = asignal * adamping
    aleft, aright pan2 asignal, p7
    ; Stereo audio output to be routed in the orchestra header.
    outleta "leftout", aleft
    outleta "rightout", aright
    endin
instr Reverberator
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ; Stereo input.
    aleftin inleta "leftin"
    arightin inleta "rightin"

```

```

idelay    =          p4
icutoff   =          p5
aleft, aright reverbsc      aleftin, arightin, idelay, icutoff
; Stereo output.
outleta   "leftout", aleft
outleta   "rightout", aright
endin

instr Compressor
; Stereo input.
aleftin  inleta   "leftin"
arightin inleta   "rightin"
kthreshold =      25000
icompl   =        0.5
icompl2  =        0.763
irtime   =        0.1
ifetime  =        0.1
aleftout  dam     aleftin, kthreshold, icompl, icompl2, irtime, iftime
arightout dam     arightin, kthreshold, icompl, icompl2, irtime, iftime
; Stereo output.
outleta   "leftout", aleftout
outleta   "rightout", arightout
endin

instr Soundfile
; Stereo input.
aleftin  inleta   "leftin"
arightin inleta   "rightin"
outs     aleftin, arightin
endin

</CsInstruments>
<CsScore>

; It is not necessary to activate "effects" or create f-tables in the score!
; Overlapping notes create new instances of instruments with proper connections.

i "SimpleSine" 1 5 60 85
i "SimpleSine" 2 5 64 80
i "Moogy"     3 5 67 75
i "Moogy"     4 5 71 70
; 1 extra second after the performance
e 1

</CsScore>
</CsoundSynthesizer>

```

## Credits

By: Michael Gogins 2009

# ampdb

ampdb — Returns the amplitude equivalent of the decibel value x.

## Description

Returns the amplitude equivalent of the decibel value x. Thus:

- 60 dB = 1000
- 66 dB = 1995.262
- 72 dB = 3891.07
- 78 dB = 7943.279
- 84 dB = 15848.926
- 90 dB = 31622.764

## Syntax

`ampdb(x)` (no rate restriction)

## Examples

Here is an example of the ampdb opcode. It uses the file `ampdb.csd` [examples/ampdb.csd].

### Example 51. Example of the ampdb opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o ampdb.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

instr 1

idb = p4
iamp = ampdb(idb)
asig oscil iamp, 220
print iamp
outs asig, asig
endin
```

```
</CsInstruments>
<CsScore>
i 1 0 1 50
i 1 + 1 90
i 1 + 1 68
i 1 + 1 80

e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like:

```
instr 1: iamp = 316.228
instr 1: iamp = 31622.763
instr 1: iamp = 2511.886
instr 1: iamp = 9999.996
```

## See Also

*ampdbfs, db, dbamp, dbfsamp*

# ampdbfs

ampdbfs — Returns the amplitude equivalent (in 16-bit signed integer scale) of the full scale decibel (dB FS) value  $x$ .

## Description

Returns the amplitude equivalent of the full scale decibel (dB FS) value  $x$ . The logarithmic full scale decibel values will be converted to linear 16-bit signed integer values from  $-32,768$  to  $+32,767$ .

## Syntax

`ampdbfs(x)` (no rate restriction)

## Examples

Here is an example of the ampdbfs opcode. It uses the file `ampdbfs.csd` [examples/ampdbfs.csd].

### Example 52. Example of the ampdbfs opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o ampdbfs.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

instr 1

idb = p4
iamp = ampdbfs(idb)
asig oscil iamp, 220, 1
print iamp
outs asig, asig
endin

</CsInstruments>
<CsScore>
;sine wave.
f 1 0 16384 10 1

i 1 0 1 -1
i 1 + 1 -5
i 1 + 1 -6
i 1 + 1 -20
```

e

```
</CsScore>  
</CsoundSynthesizer>
```

Its output should include lines like:

```
instr 1: iamp = 29204.511  
instr 1: iamp = 18426.801  
instr 1: iamp = 16422.904  
instr 1: iamp = 3276.800
```

## See Also

*ampdb*, *dbamp*, *dbfsamp*, *Odbfs*

New in Csound version 4.10

# ampmidi

ampmidi — Get the velocity of the current MIDI event.

## Description

Get the velocity of the current MIDI event.

## Syntax

```
iamp ampmidi iscal [, ifn]
```

## Initialization

*iscal* -- i-time scaling factor

*ifn* (optional, default=0) -- function table number of a normalized translation table, by which the incoming value is first interpreted. The default value is 0, denoting no translation.

## Performance

Get the velocity of the current MIDI event, optionally pass it through a normalized translation table, and return an amplitude value in the range 0 - *iscal*.

## Examples

Here is an example of the ampmidi opcode. It uses the file *ampmidi.csd* [examples/ampmidi.csd].

### Example 53. Example of the ampmidi opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
;-o ampmidi.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;expects MIDI note inputs on channel 1

iamp ampmidi 1 ; scale amplitude between 0 and 1
asig oscil iamp, 220, 1
print iamp
outs asig, asig
```

```
endin

</CsInstruments>
<CsScore>
;Dummy f-table for 1 minute
f 0 60
;sine wave.
f 1 0 16384 10 1

e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*aftouch, cpsmidi, cpsmidib, midictrl, notnum, octmidi, octmidib, pchbend, pchmidi, pchmidib, veloc*

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

# ampmidid

ampmidid — Musically map MIDI velocity to peak amplitude within a specified dynamic range in decibels.

## Description

Musically map MIDI velocity to peak amplitude within a specified dynamic range in decibels.

## Syntax

```
iamplitude ampmidid ivelocity, idecibels  
kamplitude ampmidid kvelocity, idecibels
```

## Initialization

*iamplitude* -- Amplitude.

*ivelocity* -- MIDI velocity number, ranging from 0 through 127.

*idecibels* -- Desired dynamic range in decibels.

## Performance

*kamplitude* -- Amplitude.

*kvelocity* -- MIDI velocity number, ranging from 0 through 127.

Musically map MIDI velocity to peak amplitude within a specified dynamic range in decibels:  $a = (m * v + b)^2$ , where  $a$  = amplitude,  $v$  = MIDI velocity,  $r = 10^{(R / 20)}$ ,  $b = 127 / (126 * \sqrt{r}) - 1 / 126$ ,  $m = (1 - b) / 127$ , and  $R$  = specified dynamic range in decibels. See Roger Dannenberg, "The Interpretation of MIDI Velocity," in Georg Essl and Ichiro Fujinaga (Eds.), Proceedings of the 2006 International Computer Music Conference, November 6-11, 2006 (San Francisco: The International Computer Music Association), pp. 193-196.

## Examples

Here is an example of the ampmidid opcode. It uses the file *ampmidid.csd* [examples/ampmidid.csd].

### Example 54. Example of the ampmidid opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
;;RT audio out, note=p4 and velocity=p5  
-odac --midi-key=4 --midi-velocity-amp=5  
;-iadc ;;uncomment -iadc if RT audio input is needed too  
; For Non-realtime ouput leave only the line below:
```

```
; -o ampmidid.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

massign 0, 1 ;assign all midi to instr. 1

instr 1

isine ftgenonce 0, 0, 4096, 10, 1 ;sine wave

ihz = cpsmidinn(p4)
ivelocity = p5
idb ampmidid ivelocity, 20 ;map to dynamic range of 20 dB.
idb = idb + 60 ;limit range to 60 to 80 decibels
iamplitude = ampdb(idb) ;loudness in dB to signal amplitude

a1 oscili iamplitude, ihz, isine
aenv madsr 0.05, 0.1, 0.5, 0.2
asig = a1 * aenv
outs asig, asig

endin

</CsInstruments>
<CsScore>
; note velocity
i 1 0 2 61 100
i 1 + 2 65 10
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*aftouch, cpsmidi, cpsmidib, midictrl, notnum, octmidi, octmidib, pchbend, pchmidi, pchmidib, veloc*

## Credits

Author: Michael Gogins  
2006

## areson

`areson` — A notch filter whose transfer functions are the complements of the `reson` opcode.

### Description

A notch filter whose transfer functions are the complements of the `reson` opcode.

### Syntax

```
ares areson asig, kcf, kbw [, iscl] [, iskip]
ares areson asig, acf, kbw [, iscl] [, iskip]
ares areson asig, kcf, abw [, iscl] [, iskip]
ares areson asig, acf, abw [, iscl] [, iskip]
```

### Initialization

`iscl` (optional, default=0) -- coded scaling factor for resonators. A value of 1 signifies a peak response factor of 1, i.e. all frequencies other than `kcf/acf` are attenuated in accordance with the (normalized) response curve. A value of 2 raises the response factor so that its overall RMS value equals 1. (This intended equalization of input and output power assumes all frequencies are physically present; hence it is most applicable to white noise.) A zero value signifies no scaling of the signal, leaving that to some later adjustment (see *balance*). The default value is 0.

`iskip` (optional, default=0) -- initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

### Performance

`ares` -- the output signal at audio rate.

`asig` -- the input signal at audio rate.

`kcf/acf` -- the center frequency of the filter, or frequency position of the peak response.

`kbw/abw` -- bandwidth of the filter (the Hz difference between the upper and lower half-power points).

`areson` is a filter whose transfer functions is the complement of `reson`. Thus `areson` is a notch filter whose transfer functions represents the “filtered out” aspects of their complements. However, power scaling is not normalized in `areson` but remains the true complement of the corresponding unit. Thus an audio signal, filtered by parallel matching `reson` and `areson` units, would under addition simply reconstruct the original spectrum.

This property is particularly useful for controlled mixing of different sources (see *lpreson*). Complex response curves such as those with multiple peaks can be obtained by using a bank of suitable filters in series. (The resultant response is the product of the component responses.) In such cases, the combined attenuation may result in a serious loss of signal power, but this can be regained by the use of *balance*.



#### Warning

When used with `iscl` this opcode is not a notch filter but similar to `reson`.

## Examples

Here is an example of the `areson` opcode. It uses the file `areson.csd` [examples/areson.csd].

### Example 55. Example of the `areson` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o areson.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; unfiltered noise

asig rand 0.5 ; white noise signal.
outs asig, asig
endin

instr 2 ; filtered noise

kcf init 1000
kbw init 100
asig rand 0.5
afil areson asig, kcf, kbw
afil balance afil, asig ; afil = very loud
outs afil, afil
endin

</CsInstruments>
<CsScore>

i 1 0 2
i 2 2 2
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*aresonk, atone, atonek, port, portk, reson, resonk, tone, tonek*

## Credits

Audio rate parameters introduced in version 6.02

October 2013.

# aresonk

aresonk — A notch filter whose transfer functions are the complements of the reson opcode.

## Description

A notch filter whose transfer functions are the complements of the reson opcode.

## Syntax

```
kres aresonk ksig, kcf, kbw [, iscl] [, iskip]
```

## Initialization

*iscl* (optional, default=0) -- coded scaling factor for resonators. A value of 1 signifies a peak response factor of 1, i.e. all frequencies other than *kcf* are attenuated in accordance with the (normalized) response curve. A value of 2 raises the response factor so that its overall RMS value equals 1. (This intended equalization of input and output power assumes all frequencies are physically present; hence it is most applicable to white noise.) A zero value signifies no scaling of the signal, leaving that to some later adjustment (see *balance*). The default value is 0.

*iskip* (optional, default=0) -- initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*kres* -- the output signal at control-rate.

*ksig* -- the input signal at control-rate.

*kcf* -- the center frequency of the filter, or frequency position of the peak response.

*kbw* -- bandwidth of the filter (the Hz difference between the upper and lower half-power points).

*aresonk* is a filter whose transfer functions is the complement of *resonk*. Thus *aresonk* is a notch filter whose transfer functions represents the “filtered out” aspects of their complements. However, power scaling is not normalized in *aresonk* but remains the true complement of the corresponding unit.

## Examples

Here is an example of the aresonk opcode. It uses the file *aresonk.csd* [examples/aresonk.csd].

### Example 56. Example of the aresonk opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform
```

```
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o aresonk.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gisin ftgen 0, 0, 2^10, 10, 1

instr 1

ksig randomh 400, 1800, 150
aout poscil .2, 1000+ksig, gisin
outs aout, aout
endin

instr 2

ksig randomh 400, 1800, 150
kbw line 1, p3, 600 ; vary bandwidth
ksig aresonk ksig, 800, kbw
aout poscil .2, 1000+ksig, gisin
outs aout, aout
endin

</CsInstruments>
<CsScore>
i 1 0 5
i 2 5.5 5
e
</CsScore>
</CsSoundSynthesizer>
```

## See Also

*areson, atone, atonek, port, portk, reson, resonk, tone, tonek*

# atone

*atone* — A hi-pass filter whose transfer functions are the complements of the *tone* opcode.

## Description

A hi-pass filter whose transfer functions are the complements of the *tone* opcode.

## Syntax

```
ares atone asig, khp [, iskip]
```

## Initialization

*iskip* (optional, default=0) -- initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*ares* -- the output signal at audio rate.

*asig* -- the input signal at audio rate.

*khp* -- the response curve's half-power point, in Hertz. Half power is defined as peak power / root 2.

*atone* is a filter whose transfer functions is the complement of *tone*. *atone* is thus a form of high-pass filter whose transfer functions represent the “filtered out” aspects of their complements. However, power scaling is not normalized in *atone* but remains the true complement of the corresponding unit. Thus an audio signal, filtered by parallel matching *tone* and *atone* units, would under addition simply reconstruct the original spectrum.

This property is particularly useful for controlled mixing of different sources (see *lpreson*). Complex response curves such as those with multiple peaks can be obtained by using a bank of suitable filters in series. (The resultant response is the product of the component responses.) In such cases, the combined attenuation may result in a serious loss of signal power, but this can be regained by the use of *balance*.

## Examples

Here is an example of the *atone* opcode. It uses the file *atone.csd* [examples/atone.csd].

### Example 57. Example of the *atone* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsoundOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
```

```
; For Non-realtime ouput leave only the line below:
; -o atone.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;white noise

asig rand 1
outs asig, asig

endin

instr 2 ;filtered noise

asig rand 1
khp init 4000
asig atone asig, khp
outs asig, asig

endin

</CsInstruments>
<CsScore>

i 1 0 2
i 2 2 2
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*areson, aresonk, atonek, port, portk, reson, resonk, tone, tonek*

# atonek

atonek — A hi-pass filter whose transfer functions are the complements of the *tonek* opcode.

## Description

A hi-pass filter whose transfer functions are the complements of the *tonek* opcode.

## Syntax

```
kres atonek ksig, khp [, iskip]
```

## Initialization

*iskip* (optional, default=0) -- initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*kres* -- the output signal at control-rate.

*ksig* -- the input signal at control-rate.

*khp* -- the response curve's half-power point, in Hertz. Half power is defined as peak power / root 2.

*atonek* is a filter whose transfer functions is the complement of *tonek*. *atonek* is thus a form of high-pass filter whose transfer functions represent the “filtered out” aspects of their complements. However, power scaling is not normalized in *atonek* but remains the true complement of the corresponding unit.

## Examples

Here is an example of the *atonek* opcode. It uses the file *atonek.csd* [examples/atonek.csd].

### Example 58. Example of the *atonek* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o atonek.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
```

```
gisin ftgen 0, 0, 2^10, 10, 1

instr 1

ksig randomh 400, 1800, 150
aout poscil .2, 1000+ksig, gisin
outs aout, aout
endin

instr 2

ksig randomh 400, 1800, 150
khp line 1, p3, 400 ;vary high-pass
ksig atonek ksig, khp
aout poscil .2, 1000+ksig, gisin
outs aout, aout
endin

</CsInstruments>
<CsScore>
i 1 0 5
i 2 5.5 5
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*areson, aresonk, atone, port, portk, reson, resonk, tone, tonek*

## Credits

Author: Robin Whittle  
Australia  
May 1997

# atonex

atonex — Emulates a stack of filters using the atone opcode.

## Description

*atonex* is equivalent to a filter consisting of more layers of *atone* with the same arguments, serially connected. Using a stack of a larger number of filters allows a sharper cutoff. They are faster than using a larger number instances in a Csound orchestra of the old opcodes, because only one initialization and k-cycle are needed at time and the audio loop falls entirely inside the cache memory of processor.

## Syntax

```
ares atonex asig, khp [, inumlayer] [, iskip]
```

```
ares atonex asig, ahp [, inumlayer] [, iskip]
```

## Initialization

*inumlayer* (optional) -- number of elements in the filter stack. Default value is 4.

*iskip* (optional, default=0) -- initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*asig* -- input signal

*khp/ahp* -- the response curve's half-power point. Half power is defined as peak power / root 2.

## Examples

Here is an example of the atonex opcode. It uses the file *atonex.csd* [examples/atonex.csd].

### Example 59. Example of the atonex opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o atonex.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; unfiltered noise
```

```
asig rand 0.7 ; white noise
outs asig, asig

endin

instr 2 ; filtered noise

asig rand 0.7
khp line 100, p3, 3000
afilt atonex asig, khp, 32

; Clip the filtered signal's amplitude to 85 dB.
a1 clip afilt, 2, ampdb(85)
outs a1, a1
endin

</CsInstruments>
<CsScore>

i 1 0 2
i 2 2 2

e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*resonx, tonex*

## Credits

Author: Gabriel Maldonado (adapted by John ffitich)  
Italy

New in Csound version 3.49

Audio rate parameters introduced in version 6.02

October 2013.

# ATSadd

ATSadd — uses the data from an ATS analysis file to perform additive synthesis.

## Description

*ATSadd* reads from an ATS analysis file and uses the data to perform additive synthesis using an internal array of interpolating oscillators.

## Syntax

```
ar ATSadd ktimepnt, kfmod, iatsfile, ifn, ipartialsl, ipartialoffset, \  
    ipartialincr, igatefn]
```

## Initialization

*iatsfile* – the ATS number (n in ats.n) or the name in quotes of the analysis file made using *ATS* [<http://www-ccrma.stanford.edu/~juan/ATS.html>].

*ifn* – table number of a stored function containing a sine wave for *ATSadd* and a cosine for *ATSaddnz* (see examples below for more info)

*ipartialsl* – number of partials that will be used in the resynthesis (the noise has a maximum of 25 bands)

*ipartialoffset* (optional) – is the first partial used (defaults to 0).

*ipartialincr* (optional) – sets an increment by which these synthesis opcodes counts up from *ipartialoffset* for ibins components in the re-synthesis (defaults to 1).

*igatefn* (optional) – is the number of a stored function which will be applied to the amplitudes of the analysis bins before resynthesis takes place. If *igatefn* is greater than 0 the amplitudes of each bin will be scaled by *igatefn* through a simple mapping process. First, the amplitudes of all of the bins in all of the frames in the entire analysis file are compared to determine the maximum amplitude value. This value is then used to create normalized amplitudes as indices into the stored function *igatefn*. The maximum amplitude will map to the last point in the function. An amplitude of 0 will map to the first point in the function. Values between 0 and 1 will map accordingly to points along the function table. See the examples below.

## Performance

*ktimepnt* – The time pointer in seconds used to index the ATS file. Used for *ATSadd* exactly the same as for *pvoc*.

*ATSadd* and *ATSaddnz* are based on *pvadd* by Richard Karpen and use files created by Juan Pampin's *ATS* (*Analysis - Transformation - Synthesis* [<http://www-ccrma.stanford.edu/~juan/ATS.html>]).

*kfmod* – A control-rate transposition factor: a value of 1 incurs no transposition, 1.5 transposes up a perfect fifth, and .5 down an octave. Used for *ATSadd* exactly the same as for *pvoc*.

*ATSadd* reads from an ATS analysis file and uses the data to perform additive synthesis using an internal array of interpolating oscillators. The user supplies the wave table (usually one period of a sine wave), and can choose which analysis partials will be used in the re-synthesis.

## Examples

```

ktime line 0, p3, 2.5
asig ATSadd ktime, 1, "clarinet.ats", 1, 20, 2

```

In the example above, *ipartials* is 20 and *ipartialoffset* is 2. This will synthesize the 3rd thru 22nd partials in the "clarinet.ats" analysis file. *kfmod* is 1 so there will be no pitch transformation. Since the *ktimepnt* envelope moves from 0 to 2.5 over the duration of the note, the analysis file will be read from 0 to 2.5 seconds of the original duration of the analysis over the duration of the csound note, this way we can change the duration independent of the pitch.

## Examples

Here is a complete example of the ATSadd opcode. It uses the file *ATSadd.csd* [examples/ATSadd.csd].

### Example 60. Example of the ATSadd opcode.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc for RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o ATSadd.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; "beats.ats" is created by atsa

ktime line 0, p3, 2
asig ATSadd ktime, 1, "beats.ats", 1, 20, 0, 2
outs asig*3, asig*3 ;amplify

endin

</CsInstruments>
<CsScore>
;sine wave.
f 1 0 16384 10 1

i 1 0 2
e

</CsScore>
</CsoundSynthesizer>

```

In the above example we synthesize 20 partials as in example 1 except this time we're using a *ipartialoffset* of 0 and *ipartialincr* of 2, which means that we'll start from the first partial and synthesize 20 partials total, skipping every other one (ie. partial 1, 3, 5,...).

Here is another example of the ATSadd opcode. It uses the file *ATSadd-2.csd* [examples/ATSadd-2.csd].

### Example 61. Example 2 of the ATSadd opcode.

```

<CsoundSynthesizer>
<CsOptions>
-odac -d -ml
</CsOptions>
<CsInstruments>
;example by joachim heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine    ftgen    0, 0, 1024, 10, 1

  instr AllTheTones
Sfile    =        "fox.ats"
          prints   "Resynthesizing with all the tones.\n"
iDur     =        ATSinfo Sfile, 7
p3       =        iDur
iNumParts = ATSinfo Sfile, 3
          prints   "Overall number of partials = %d\n", iNumParts
ktime    =        line 0, iDur, iDur
asig     =        ATSadd ktime, 1, Sfile, giSine, iNumParts
          outs     asig, asig

;start next instr
          event_i   "i", "TonesInBandsOfTen", iDur+1, iDur, 0, iNumParts
  endin

  instr TonesInBandsOfTen
Sfile    =        "fox.ats"
iOffset  =        p4 ;start at this partial
iNumParts = p5 ;overall number of partials
          prints   "Resynthesizing with partials %d - %d.\n", iOffset+1, iOffset+10
ktime    =        line 0, p3, p3
asig     =        ATSadd ktime, 1, Sfile, giSine, 10, iOffset
          outs     asig, asig

;start next instance until there are enough partials left
  if iOffset+20 < iNumParts then
    event_i   "i", "TonesInBandsOfTen", p3+1, p3, iOffset+10, iNumParts
  else
    event_i   "i", "End", p3, 1
  endif
  endin

  instr End
          exitnow
  endin
</CsInstruments>
<CsScore>
i "AllTheTones" 0 1
e 999
</CsScore>
</CsoundSynthesizer>

```

## See also

*ATSread, ATSreadnz, ATSinfo, ATSbufread, ATScross, ATSinterpread, ATSpartialtap, ATSaddnz, ATSSinnoi*

## Credits

Author: Alex Norman

Seattle, Washington  
2004

# ATSaddnz

ATSaddnz — uses the data from an ATS analysis file to perform noise resynthesis.

## Description

*ATSaddnz* reads from an ATS analysis file and uses the data to perform additive synthesis using a modified *randi* function.

## Syntax

```
ar ATSaddnz ktimepnt, iatsfile, ibands[, ibandoffset, ibandincr]
```

## Initialization

*iatsfile* – the ATS number (n in ats.n) or the name in quotes of the analysis file made using *ATS* [<http://www-ccrma.stanford.edu/~juan/ATS.html>].

*ibands* – number of noise bands that will be used in the resynthesis (the noise has a maximum of 25 bands)

*ibandoffset* (optional) – is the first noise band used (defaults to 0).

*ibandincr* (optional) – sets an increment by which these synthesis opcodes counts up from *ibandoffset* for *ibins* components in the re-synthesis (defaults to 1).

## Performance

*ktimepnt* – The time pointer in seconds used to index the ATS file. Used for *ATSaddnz* exactly the same as for *pvoc* and *ATSadd*.

*ATSaddnz* and *ATSadd* are based on *pvadd* by Richard Karpen and use files created by Juan Pampin's *ATS* (*Analysis - Transformation - Synthesis* [<http://www-ccrma.stanford.edu/~juan/ATS.html>]).

*ATSaddnz* also reads from an ATS file but it resynthesizes the noise from noise energy data contained in the ATS file. It uses a modified *randi* function to create band limited noise and modulates that with a cosine wave, to synthesize a user specified selection of frequency bands. Modulating the noise is required to put the band limited noise in the correct place in the frequency spectrum.

## Examples

```
ktime line 0, p3, 2.5
asig ATSaddnz ktime, "clarinet.ats", 25
```

In the example above we're synthesizing all 25 noise bands from the data contained in the ATS analysis file called "clarinet.ats".

## Examples

Here is a complete example of the *ATSaddnz* opcode. It uses the file *ATSaddnz.csd* [examples/*ATSaddnz.csd*].

**Example 62. Example of the ATSaddnz opcode.**

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc for RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o ATSaddnzwav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; "beats.ats" is created by atsa

ktime line 0, p3, 2
asig ATSaddnz ktime, "cage.ats", 1, 24
outs asig*10, asig*10 ;amplify
endin

</CsInstruments>
<CsScore>

i 1 0 2
e

</CsScore>
</CsoundSynthesizer>

```

Here we synthesize only the 25th noise band (*ibandoffset* of 24 and *ibands* of 1).

Here is another example of the ATSaddnz opcode. It uses the file *ATSaddnz-2.csd* [examples/ATSaddnz-2.csd].

**Example 63. Example 2 of the ATSaddnz opcode.**

```

<CsoundSynthesizer>
<CsOptions>
-odac -d -ml
</CsOptions>
<CsInstruments>
;example by joachim heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr AllTheNoise
Sfile = "fox.ats"
prints "Resynthesizing with all the noise.\n"
iDur ATSinfo Sfile, 7
p3 = iDur
ktime line 0, iDur, iDur
asig ATSaddnz ktime, Sfile, 25
outs asig, asig

;start next instr
event_i "i", "NoiseInBandsOfFive", iDur+1, 1, 0

```

```
endin

instr NoiseInBandsOfFive
Sfile = "fox.ats"
prints "Resynthesizing with noise bands %d - %d.\n", p4, p4+5
iDur ATInfo Sfile, 7
p3 = iDur
ktime line 0, iDur, iDur
asig ATSaddnz ktime, Sfile, 5, p4
outs asig, asig

;start next instr
if p4 < 20 then
event_i "i", "NoiseInBandsOfFive", iDur+1, 1, p4+5
endif
endin
</CsInstruments>
<CsScore>
i "AllTheNoise" 0 1
e 25
</CsScore>
</CsoundSynthesizer>
```

## See also

*ATSread, ATSreadnz, ATInfo, ATBufread, ATScross, ATSinterpread, ATSpartialtap, ATSaddnz, ATSSin-noi*

## Credits

Author: Alex Norman  
Seattle, Washington  
2004

# ATSbufread

*ATSbufread* — reads data from and ATS data file and stores it in an internal data table of frequency, amplitude pairs.

## Description

*ATSbufread* reads data from and ATS data file and stores it in an internal data table of frequency, amplitude pairs.

## Syntax

```
ATSbufread ktimepnt, kfmod, iatsfile, ipartial[, ipartialoffset, \  
            ipartialincr]
```

## Initialization

*iatsfile* – the ATS number (n in ats.n) or the name in quotes of the analysis file made using *ATS* [<http://www-ccrma.stanford.edu/~juan/ATS.html>].

*ipartial*s – number of partials that will be used in the resynthesis (the noise has a maximum of 25 bands)

*ipartialoffset* (optional) – is the first partial used (defaults to 0).

*ipartialincr* (optional) – sets an increment by which these synthesis opcodes counts up from *ipartialoffset* for ibins components in the re-synthesis (defaults to 1).

## Performance

*ktimepnt* – The time pointer in seconds used to index the ATS file. Used for *ATSbufread* exactly the same as for *pvoc*.

*kfmod* – an input for performing pitch transposition or frequency modulation on all of the synthesized partials, if no fm or pitch change is desired then use a 1 for this value.

*ATSbufread* is based on *pvbufread* by Richard Karpen. *ATScross*, *ATSinterpread* and *ATSpartialtap* are all dependent on *ATSbufread* just as *pvcross* and *pvinterp* are on *pvbufread*. *ATSbufread* reads data from and ATS data file and stores it in an internal data table of frequency, amplitude pairs. The data stored by an *ATSbufread* can only be accessed by other unit generators, and therefore, due to the architecture of Csound, an *ATSbufread* must come before (but not necessarily directly) any dependent unit generator. Besides the fact that *ATSbufread* doesn't output any data directly, it works almost exactly as *ATSadd*. The ugen uses a time pointer (*ktimepnt*) to index the data in time, *ipartial*s, *ipartialoffset* and *ipartialincr* to select which partials to store in the table and *kfmod* to scale partials in frequency.

## Examples

Here is an example of the *ATSbufread* opcode. It uses the file *ATSbufread.csd* [examples/ATSbufread.csd].

**Example 64. Example of the *ATSbufread* opcode.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc for RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o ATSBufread.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; "beats.ats" and "fox.ats" are created by atsa

ktime line 0, p3, 4
ktime2 line 0, p3, 4
kline expseg 0.001, .3, 1, p3-.3, 1
kline2 expseg 0.001, p3, 3
      ATSBufread ktime2, 1, "fox.ats", 20
aout ATScross ktime, 2, "beats.ats", 1, kline, 0.001 * (4 - kline2), 180
outs aout*2, aout*2

endin

</CsInstruments>
<CsScore>
; sine wave.
f 1 0 16384 10 1

i 1 0 4
e
</CsScore>
</CsoundSynthesizer>
```

See also the examples for *ATScross*, *ATSinterpread* and *ATSpartialtap*

## See also

*ATSread*, *ATSreadnz*, *ATSinfo*, *ATSsinnoi*, *ATScross*, *ATSinterpread*, *ATSpartialtap*, *ATSadd*, *ATSaddnz*

## Credits

Author: Alex Norman  
Seattle, Washington  
2004

# ATScross

ATScross — perform cross synthesis from ATS analysis files.

## Description

*ATScross* uses data from an ATS analysis file and data from an *ATSbufread* to perform cross synthesis.

## Syntax

```
ar ATScross ktimepnt, kfmmod, iatsfile, ifn, kmylev, kbuflev, ipartials \  
    [, ipartialoffset, ipartialincr]
```

## Initialization

*iatsfile* – integer or character-string denoting a control-file derived from ATS analysis of an audio signal. An integer denotes the suffix of a file *ATS.m*; a character-string (in double quotes) gives a filename, optionally a full pathname. If not full-path, the file is sought first in the current directory, then in the one given by the environment variable *SADIR* (if defined).

*ifn* – table number of a stored function containing a sine wave.

*ipartials* – number of partials that will be used in the resynthesis

*ipartialoffset* (optional) – is the first partial used (defaults to 0).

*ipartialincr* (optional) – sets an increment by which these synthesis opcodes counts up from *ipartialoffset* for ibins components in the re-synthesis (defaults to 1).

## Performance

*ktimepnt* – The time pointer in seconds used to index the ATS file. Used for *ATScross* exactly the same as for *pvoc*.

*kfmmod* – an input for performing pitch transposition or frequency modulation on all of the synthesized partials, if no fm or pitch change is desired then use a 1 for this value.

*kmylev* - scales the *ATScross* component of the frequency spectrum applied to the partials from the ATS file indicated by the *ATScross* opcode. The frequency spectrum information comes from the *ATScross* ATS file. A value of 1 (and 0 for *kbuflev*) gives the same results as *ATSadd*.

*kbuflev* - scales the *ATSbufread* component of the frequency spectrum applied to the partials from the ATS file indicated by the *ATScross* opcode. The frequency spectrum information comes from the *ATSbufread* ATS file. A value of 1 (and 0 for *kmylev*) results in partials that have frequency information from the ATS file given by the *ATScross*, but amplitudes imposed by data from the ATS file given by *ATSbufread*.

*ATScross* uses data from an ATS analysis file (indicated by *iatsfile*) and data from an *ATSbufread* to perform cross synthesis. *ATScross* uses *ktimepnt*, *kfmmod*, *ipartials*, *ipartialoffset* and *ipartialincr* just like *ATSadd*. *ATScross* synthesizes a sine-wave for each partial selected by the user and uses the frequency of that partial (after scaling in frequency by *kfmmod*) to index the table created by *ATSbufread*. Interpolation is used to get in-between values. *ATScross* uses the sum of the amplitude data from its ATS file (scaled by *kmylev*) and the amplitude data gained from an *ATSbufread* (scaled by *kbuflev*) to scale the amplitude of

each partial it synthesizes. Setting *kmylev* to one and *kbuflev* to zero will make *ATScross* act exactly like *ATSadd*. Setting *kmylev* to zero and *kbuflev* to one will produce a sound that has all the partials selected by the *ATScross* ugen, but with amplitudes taken from an *ATSbufread*. The time pointers of the *ATSbufread* and *ATScross* do not need to be the same.

## Examples

Here is an example of the *ATScross* opcode. It uses the file *ATScross.csd* [examples/ATScross.csd].

### Example 65. Example of the *ATScross* opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc for RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o ATScross.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; "beats.ats" and "fox.ats" are created by atsa

ktime line 0, p3, 4
ktime2 line 0, p3, 4
kline expseg 0.001, .3, 1, p3-.3, 1
kline2 expseg 0.001, p3, 3
    ATScross ktime2, 1, "fox.ats", 20
out ATScross ktime, 2, "beats.ats", 1, kline, 0.001 * (4 - kline2), 180
outs out*2, out*2

endin

</CsInstruments>
<CsScore>
; sine wave.
f 1 0 16384 10 1

i 1 0 4
e
</CsScore>
</CsoundSynthesizer>
```

This example performs cross synthesis using two ATS-files, "fox.ats" and "beats.ats". The result of this will be a sound that starts out with the shape (in frequency) of fox.ats, and ends with the shape of beats.ats. All the sine-wave frequencies come from beats.ats. The *kbuflev* value is scaled because the energy produced by applying fox.ats's frequency spectrum to beats.ats's partials is very large. Notice also that the time pointers of the *ATSbufread* (fox.ats) and *ATScross* (beats.ats) need not have the same value, this way you can read through the two ATS files at different rates.

## See also

*ATSread*, *ATSreadnz*, *ATSinfo*, *ATSsinnoi*, *ATSbufread*, *ATSinterpread*, *ATSpartialtap*, *ATSadd*, *ATSaddnz*

## Credits

Author: Alex Norman  
Seattle, Washington  
2004

# ATInfo

ATInfo — reads data out of the header of an ATS file.

## Description

*atsinfo* reads data out of the header of an ATS file.

## Syntax

```
idata ATInfo iatsfile, ilocation
```

## Initialization

*iatsfile* – the ATS number (n in ats.n) or the name in quotes of the analysis file made using *ATS* [<http://www-ccrma.stanford.edu/~juan/ATS.html>].

*ilocation* – indicates which location in the header file to return. The data in the header gives information about the data contained in the rest of the ATS file. The possible values for *ilocation* are given in the following list:

- 0 - Sample rate (Hz)
- 1 - Frame Size (samples)
- 2 - Window Size (samples)
- 3 - Number of Partialals
- 4 - Number of Frames
- 5 - Maximum Amplitude
- 6 - Maximum Frequency (Hz)
- 7 - Duration (seconds)
- 8 - ATS file Type

## Performance

Macros can really improve the legibility of your csound code, I've provided my Macro Definitions below:

```
#define ATS_SAMP_RATE #0#  
#define ATS_FRAME_SZ #1#  
#define ATS_WIN_SZ #2#  
#define ATS_N_PARTIALS #3#  
#define ATS_N_FRAMES #4#  
#define ATS_AMP_MAX #5#  
#define ATS_FREQ_MAX #6#  
#define ATS_DUR #7#  
#define ATS_TYPE #8#
```

*ATInfo* can be useful for writing generic instruments that will work with many ATS files, even if they have different lengths and different numbers of partials etc. Example 2 is a simple application of this.

## Examples

Here is an example of the *ATSinfo* opcode. It uses the file *ATSinfo.csd* [examples/ATSinfo.csd].

### Example 66. Example of the *ATSinfo* opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

instr 1 ; "fox.ats" is created by atsa

inum_partials ATSinfo "fox.ats", 3
  print inum_partials

endin

</CsInstruments>
<CsScore>
i 1 0 0
e

</CsScore>
</CsoundSynthesizer>
```

In the example above we use *ATSinfo* to retrieve the number of partials in the ATS file

## Other examples

1.
 

```
imax_freq      ATSinfo "cl.ats", $ATS_FREQ_MAX
```

In the example above we get the maximum frequency value from the ATS file "cl.ats" and store it in *imax\_freq*. We use the Csound Macro (defined above) *\$ATS\_FREQ\_MAX*, which is equivalent to the number 6.

2.
 

```
i_npartials    ATSinfo p4, $ATS_N_PARTIALS
i_dur          ATSinfo p4, $ATS_DUR
ktimepnt      line    0, p3, i_dur
aout          ATSadd ktimepnt, 1, p4, 1, i_npartials
```

In the example above we use *ATSinfo* to retrieve the duration and number of partials in the ATS file indicated by *p4*. With this info we synthesize the partials using *atsadd*. Since the duration and number of partials are not "hard-coded" we can use this code with any ATS file.

Here is another example of the *ATSinfo* opcode. It uses the file *ATSinfo-2.csd* [examples/ATSinfo-2.csd].

### Example 67. Example 2 of the *ATSinfo* opcode.

```
<CsoundSynthesizer>
```

```
<CsOptions>
; Select audio/midi flags here according to platform
-n      ;;no audio out
</CsOptions>
<CsInstruments>
;example by joachim heintz
sr = 44100
kmps = 32
nchnls = 2
odbfs = 1

instr 1 ; "fox.ats" has been created by ATSanal
Sfile = "fox.ats"
isr   ATInfo Sfile, 0
ifs   ATInfo Sfile, 1
iws   ATInfo Sfile, 2
inp   ATInfo Sfile, 3
inf   ATInfo Sfile, 4
ima   ATInfo Sfile, 5
imf   ATInfo Sfile, 6
id    ATInfo Sfile, 7
ift   ATInfo Sfile, 8
      prints {{
Sample rate = %d Hz
Frame Size = %d samples
Window Size = %d samples
Number of Partial = %d
Number of Frames = %d
Maximum Amplitude = %f
Maximum Frequency = %f Hz
Duration = %f seconds
ATS file Type = %d
}}, isr, ifs, iws, inp, inf, ima, imf, id, ift
endin
</CsInstruments>
<CsScore>
i 1 0 0
</CsScore>
</CsoundSynthesizer>
```

## See also

*ATSread, ATSreadnz, ATSbufread, ATScross, ATSinterpread, ATSpartialtap, ATSadd, ATSaddnz, ATSSin-  
noi*

## Credits

Author: Alex Norman  
Seattle, Washington  
2004

# ATSinterpread

ATSinterpread — allows a user to determine the frequency envelope of any *ATSbufread*.

## Description

*ATSinterpread* allows a user to determine the frequency envelope of any *ATSbufread*.

## Syntax

```
kamp ATSinterpread kfreq
```

## Performance

*kfreq* - a frequency value (given in Hertz) used by *ATSinterpread* as an index into the table produced by an *ATSbufread*.

*ATSinterpread* takes a frequency value (*kfreq* in Hz). This frequency is used to index the data of an *ATSbufread*. The return value is an amplitude gained from the *ATSbufread* after interpolation. *ATSinterpread* allows a user to determine the frequency envelope of any *ATSbufread*. This data could be useful for a number of reasons, one might be performing cross synthesis of data from an ATS file and non ATS data.

## Examples

Here is an example of the *ATSinterpread* opcode. It uses the file *ATSinterpread.csd* [examples/ATSinterpread.csd].

### Example 68. Example of the *ATSinterpread* opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc for RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o ATSinterpread.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; "beats.ats" is created by atsa

ktime line 0, p3, 1.8
  ATSbufread ktime, 1, "beats.ats", 42
  kamp ATSinterpread p4
  aosc oscili kamp, p4, 1
  outs aosc * 25, aosc * 25

endin
```

```
</CsInstruments>
<CsScore>
; sine wave.
f 1 0 16384 10 1

i 1 0 2 100
e

</CsScore>
</CsoundSynthesizer>
```

This example shows how to use *ATSinterpread*. Here a frequency is given by the score (p4) and this frequency is given to an *ATSinterpread* (with a corresponding *ATSbufread*). The *ATSinterpread* uses this frequency to output a corresponding amplitude value, based on the atsfile given by the *ATSbufread* (beats.ats in this case). We then use that amplitude to scale a sine-wave that is synthesized with the same frequency (p4). You could extend this to include multiple sine-waves. This way you could synthesize any reasonable frequency (within the low and high frequencies of the indicated ATS file), and maintain the shape (in frequency) of the indicated atsfile (given by the *ATSbufread*).

## See also

*ATSread*, *ATSreadnz*, *ATSinfo*, *ATSsinnoi*, *ATSbufread*, *ATScross*, *ATSpartialtap*, *ATSadd*, *ATSaddnz*

## Credits

Author: Alex Norman  
Seattle, Washington  
2004

# ATSread

ATSread — reads data from an ATS file.

## Description

*ATSread* returns the amplitude (*kamp*) and frequency (*kfreq*) information of a user specified partial contained in the ATS analysis file at the time indicated by the time pointer *ctimepnt*.

## Syntax

```
kfreq, kamp ATSread ctimepnt, iatsfile, ipartial
```

## Initialization

*iatsfile* – the ATS number (n in ats.n) or the name in quotes of the analysis file made using *ATS* [<http://www-ccrma.stanford.edu/~juan/ATS.html>].

*ipartial* – the number of the analysis partial to return the frequency in Hz and amplitude.

## Performance

*kfreq*, *kamp* - outputs of the *ATSread* unit. These values represent the frequency and amplitude of a specific partial selected by the user using *ipartial*. The partials' informations are derived from an ATS analysis. *ATSread* linearly interpolates the frequency and amplitude between frames in the ATS analysis file at k-rate. The output is dependent on the data in the analysis file and the pointer *ctimepnt*.

*ctimepnt* – The time pointer in seconds used to index the ATS file. Used for *ATSread* exactly the same as for *pvoc* and *ATSadd*.

## Examples

Here is an example of the *ATSread* opcode. It uses the file *ATSread.csd* [examples/ATSread.csd].

### Example 69. Example of the *ATSread* opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o ATSread.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; "beats.ats" is created by atsa

ktime line 0, p3, 2
kfreq, kamp ATSread ktime, "beats.ats", 100
```

```

aout oscili 0.8, kfreq, 1
outs aout, aout

endin

</CsInstruments>
<CsScore>
; sine wave.
f 1 0 16384 10 1

i 1 0 2
e

</CsScore>
</CsoundSynthesizer>

```

Here we're using *ATSread* to get the 100th partial's frequency and amplitude data out of the 'beats.ats' ATS analysis file. We're using that data to drive an oscillator, but we could use it for anything else that can take a k-rate input, like the bandwidth and resonance of a filter etc.

Here is another example of the *ATSread* opcode. It uses the file *ATSread-2.csd* [examples/ATSread-2.csd].

### Example 70. Example 2 of the *ATSread* opcode.

```

<CsoundSynthesizer>
<CsOptions>
-odac -d -m128
</CsOptions>
<CsInstruments>
; example by joachim heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine ftgen 0, 0, 1024, 10, 1
gSfile = "fox.ats"
giNumParts ATSinfo gSfile, 3 ;overall number of partials
giDur ATSinfo gSfile, 7 ;duration
seed 0

instr ReadOnePartial
iPartial = p4
p3 = giDur
ktime line 0, giDur, giDur
prints "Resynthesizing partial number %d.\n", iPartial
kFq, kAmp ATSread ktime, gSfile, iPartial
kAmp port kAmp, .1 ;smooth amplitudes - still not satisfactoring
aOut poscil kAmp, kFq, giSine
aOut linen aOut, 0, p3, .01 ;anti-click
outs aOut*10, aOut*10

;start next instr: normal speed, three loops, pause between loops one second
event_i "i", "MasterRand", giDur+3, 1, 1, 3, 2
endin

instr MasterRand
;random selections of 10 partials per second, overlapping
iSpeed = p4 ;speed of reading / playing
iNumLoops = p5 ;number of loops
iPause = p6 ;length of pause between loops
prints "Resynthesizing random partials.\n"
p3 = (giDur/iSpeed+iPause) * iNumLoops
;start next instr: half speed, three loops, three seconds pause between loops

```

```

        event_i "i", "MasterArp", p3+3, 1, .5, 3, 3
;loop over duration plus pause
loop:
    timeout 0, giDur/iSpeed+iPause, play
    reinit  loop

play:
gkTime  line 0, giDur/iSpeed, giDur ;start time from 0 in each loop
kTrig   metro 10 ;10 new partials per second
        ;call subinstrument if trigger and no pause
        if kTrig == 1 && gkTime < giDur then
kPart   random 1, giNumParts+.999
        event "i", "PlayRand", 0, 1, int(kPart)
        endif
    endif

    endin

    instr MasterArp
        ;arpeggio-like reading and playing of partials
iSpeed  = p4 ;speed of reading / playing
iNumLoops = p5 ;number of loops
iPause  = p6 ;length of pause between loops
        prints "Arpeggiating partials.\n"
p3      = (giDur/iSpeed+iPause) * iNumLoops
loop:
    timeout 0, giDur/iSpeed+iPause, play
    reinit  loop

play:
gkTime  line 0, giDur/iSpeed, giDur
kArp    linseq 1, (giDur/iSpeed)/2, giNumParts, (giDur/iSpeed)/2, 1 ;arp up and down
kTrig   metro 10 ;10 new partials per second
        if kTrig == 1 && gkTime < giDur then
        event "i", "PlayArp", 0, 5, int(kArp)
        endif
    endif

;exit csound when finished
        event_i "i", "End", p3+5, 1
    endin

    instr PlayRand
iPartial = p4
kFq,kAmp ATsread gkTime, gSfile, iPartial
kAmp    port kAmp, .15 ;smooth amplitudes
aOut    poscil kAmp, kFq, giSine
aOut    linen aOut, .01, p3, .01
        outs aOut, aOut
    endin

    instr PlayArp
kCount  init 1 ;k-cycle
iPartial = p4
kFq,kAmp ATsread gkTime, gSfile, iPartial
        if kCount == 1 then ;get freq from first k-cycle
kModFq  = kFq
        ;avoid to go with 0 Hz as this blocks the mode filter
        if kModFq == 0 then
        turnoff
        endif
        endif
iVol    random -42, -12 ;db
iOffset random .01, .1 ;no too regularly ...
aImp    mpulse ampdb(iVol), p3, iOffset
iQ      random 500, 5000
aOut    mode aImp, kModFq, iQ
aOut    linen aOut, 0, p3, p3/3
        outs aOut, aOut
kCount  = 2
    endin

```

```
instr End
  exitnow
endin
</CsInstruments>
<CsScore>
i "ReadOnePartial" 0 1 10
e 999
</CsScore>
</CsoundSynthesizer>
```

## See also

*ATSreadnz*, *ATSinfo*, *ATSbufread*, *ATScross*, *ATSinterpread*, *ATSpartialtap*, *ATSadd*, *ATSaddnz*, *ATSsin-  
noi*

## Credits

Author: Alex Norman  
Seattle, Washington  
2004

# ATSreadnz

ATSreadnz — reads data from an ATS file.

## Description

*ATSreadnz* returns the energy (*kenergy*) of a user specified noise band (1-25 bands) at the time indicated by the time pointer *ktimepnt*.

## Syntax

```
kenergy ATSreadnz ktimepnt, iatsfile, iband
```

## Initialization

*iatsfile* – the ATS number (n in ats.n) or the name in quotes of the analysis file made using *ATS* [<http://www-ccrma.stanford.edu/~juan/ATS.html>].

*iband* – the number of the noise band to return the energy data.

## Performance

*kenergy* outputs the linearly interpolated energy of the noise band indicated in *iband*. The output is dependent on the data in the analysis file and the *ktimepnt*.

*ktimepnt* – The time pointer in seconds used to index the ATS file. Used for *ATSreadnz* exactly the same as for *pvoc* and *ATSadd*.

*ATSaddnz* reads from an ATS file and resynthesizes the noise from noise energy data contained in the ATS file. It uses a modified *randi* function to create band limited noise and modulates that with a user supplied wave table (one period of a cosine wave), to synthesize a user specified selection of frequency bands. Modulating the noise is required to put the band limited noise in the correct place in the frequency spectrum.

An ATS analysis differs from a *pvanal* in that ATS tracks the partials and computes the noise energy of the sound being analyzed. For more info about ATS analysis read Juan Pampin's description on the the *ATS* [<http://www-ccrma.stanford.edu/~juan/ATS.html>] web-page.

## Examples

```
ktime  line      2.5, p3, 0
kenergy ATSreadnz ktime, "clarinet.ats", 5
```

Here we are extracting the noise energy from band 5 in the 'clarinet.ats' ATS analysis file. We're actually reading backwards from 2.5 seconds to the beginning of the analysis file. We could use this to synthesize noise like this:

```
anoise randi      sqrt(kenergy), 55
aout   oscili    40000000000000000000000000000000, 455, 2
aout   =          aout * anoise
```

Function table 2 used in the oscillator is a cosine, which is needed to shift the band limited noise into the correct place in the frequency spectrum. The *randi* function creates a band of noise centered about 0 Hz that has a bandwidth of about 110 Hz; multiplying it by a cosine will shift it to be centered at 455 Hz, which is the center frequency of the 5th critical noise band. This is only an example, for synthesizing the noise you'd be better off just using *ATSaddnz* unless you want to use your own noise synthesis algorithm. Maybe you could use the noise energy for something else like applying a small amount of jitter to specific partials or for controlling something totally unrelated to the source sound?

## Examples

Here is another example of the *ATSreadnz* opcode. It uses the file *ATSreadnz.csd* [examples/ATSreadnz.csd].

### Example 71. Another example of the *ATSreadnz* opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
-iadc      ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o ATSreadnz.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; "beats.ats" is created by atsa

ktime line 0, p3, 2
kenergy ATSreadnz ktime, "beats.ats", 2
anoise randi kenergy, 500
aout oscili 0.005, 455, 1
aout = aout * anoise
outs aout, aout
endin

</CsInstruments>
<CsScore>
; cosine wave
f 1 0 16384 11 1 1

i 1 0 2
e

</CsScore>
</CsoundSynthesizer>
```

## See also

*ATSread*, *ATSinfo*, *ATSbufread*, *ATScross*, *ATSinterpread*, *ATSpartialtap*, *ATSadd*, *ATSaddnz*, *ATSsinnoi*

## Credits

Author: Alex Norman  
Seattle, Washington

2004

# ATSpartialtap

ATSpartialtap — returns a frequency, amplitude pair from an *ATSbufread* opcode.

## Description

*ATSpartialtap* takes a partial number and returns a frequency, amplitude pair. The frequency and amplitude data comes from an *ATSbufread* opcode.

## Syntax

```
kfrq, kamp ATSpartialtap ipartialnum
```

## Initialization

*ipartialnum* - indicates the partial that the *ATSpartialtap* opcode should read from an *ATSbufread*.

## Performance

*kfrq* - returns the frequency value for the requested partial.

*kamp* - returns the amplitude value for the requested partial.

*ATSpartialtap* takes a partial number and returns a frequency, amplitude pair. The frequency and amplitude data comes from an *ATSbufread* opcode. This is more restricted version of *ATSread*, since each *ATSread* opcode has its own independent time pointer, and *ATSpartialtap* is restricted to the data given by an *ATSbufread*. Its simplicity is its attractive feature.

## Examples

Here is an example of the *ATSpartialtap* opcode. It uses the file *ATSpartialtap.csd* [examples/ATSpartialtap.csd].

### Example 72. Example of the *ATSpartialtap* opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc for RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o ATSpartialtap.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; "beats.ats" is created by atsa

ktime line 0, p3, 2
```

```
    ATSbufread ktime, 1, "beats.ats", 30
    kfreq1, kam1 ATSpartialtap 5
    kfreq2, kam2 ATSpartialtap 20
    kfreq3, kam3 ATSpartialtap 30

    aout1 oscil kam1, kfreq1, 1
    aout2 oscil kam2, kfreq2, 1
    aout3 oscil kam3, kfreq3, 1
    aout = (aout1+aout2+aout3)*10 ; amplify some more
    outs aout, aout

    endin

</CsInstruments>
<CsScore>
; sine wave.
f 1 0 16384 10 1
i 1 0 2
e

</CsScore>
</CsoundSynthesizer>
```

This example here uses an *ATSpartialtap*, and an *ATSbufread* to read partials 5, 20 and 30 from 'beats.ats'. These amplitudes and frequencies could be used to re-synthesize those partials, or something all together different.

## See also

*ATSread*, *ATSreadnz*, *ATSinfo*, *ATSsinnoi*, *ATSbufread*, *ATScross*, *ATSinterpread*, *ATSadd*, *ATSaddnz*

## Credits

Author: Alex Norman  
Seattle, Washington  
2004

# ATSSinnoi

ATSSinnoi — uses the data from an ATS analysis file to perform resynthesis.

## Description

*ATSSinnoi* reads data from an ATS data file and uses the information to synthesize sines and noise together.

## Syntax

```
ar ATSSinnoi ktimepnt, ksinlev, knzlev, kfmod, iatsfile, ipartial \
    [, ipartialoffset, ipartialincr]
```

## Initialization

*iatsfile* – the ATS number (n in ats.n) or the name in quotes of the analysis file made using *ATS* [<http://www-ccrma.stanford.edu/~juan/ATS.html>].

*ipartial* – number of partials that will be used in the resynthesis (the noise has a maximum of 25 bands)

*ipartialoffset* (optional) – is the first partial used (defaults to 0).

*ipartialincr* (optional) – sets an increment by which these synthesis opcodes counts up from *ipartialoffset* for ibins components in the re-synthesis (defaults to 1).

## Performance

*ktimepnt* – The time pointer in seconds used to index the ATS file. Used for *ATSSinnoi* exactly the same as for *pvoc*.

*ksinlev* - controls the level of the sines in the *ATSSinnoi* ugen. A value of 1 gives full volume sinewaves.

*knzlev* - controls the level of the noise components in the *ATSSinnoi* ugen. A value of 1 gives full volume noise.

*kfmod* – an input for performing pitch transposition or frequency modulation on all of the synthesized partials, if no fm or pitch change is desired then use a 1 for this value.

*ATSSinnoi* reads data from an ATS data file and uses the information to synthesize sines and noise together. The noise energy for each band is distributed equally among each partial that falls in that band. Each partial is then synthesized, along with that partial's noise component. Each noise component is then modulated by the corresponding partial to be put in the correct place in the frequency spectrum. The level of the noise and the partials are individually controllable. See the *ATS* [<http://www-ccrma.stanford.edu/~juan/ATS.html>] webpage for more info about the sinnoi synthesis. An ATS analysis differs from a pvanal in that ATS tracks the partials and computes the noise energy of the sound being analyzed. For more info about ATS analysis read Juan Pampin's description on the the *ATS* [<http://www-ccrma.stanford.edu/~juan/ATS.html>] web-page.

## Examples

```
ktime   line      0, p3, 2.5
asig    ATSSinnoi ktime, 1, 1, 1, "beats.ats", 42
```

Here we synthesize both the noise and the sinewaves (all 42 partials) contained in "beats.ats" together. The relative volumes of the noise and the partials are unaltered (each set to 1).

Here is another example of the ATSSinnoi opcode. It uses the file *ATSSinnoi.csd* [examples/ATSSinnoi.csd].

### Example 73. Example of the ATSSinnoi opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o ATSSinnoi.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; "beats.ats" is created by atsa

ktime line 0, p3, 2
knzfade expon 0.001, p3, 2
aout ATSSinnoi ktime, 1, knzfade, 1, "beats.ats", 150
outs aout*2, aout*2 ;amplify some more
endin

</CsInstruments>
<CsScore>

i 1 0 2
e

</CsScore>
</CsoundSynthesizer>
```

This example here is like the other example except that we use an envelope to control *knzlev* (the noise level). The result of this will be the "beats.wav" sound that has its noise component fade in over the duration of the note.

Here is another example of the ATSSinnoi opcode. It uses the file *ATSSinnoi-2.csd* [examples/ATSSinnoi-2.csd].

### Example 74. Example 2 of the ATSSinnoi opcode.

```
<CsoundSynthesizer>
<CsOptions>
-odac -d -m128
</CsOptions>
<CsInstruments>
;example by joachim heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine ftgen 0, 0, 1024, 10, 1
gSfile = "fox.ats"
```

## Orchestra Opcodes and Operators

---

```

giNumParts  ATsinfo  gSfile, 3 ;overall number of partials
giDur       ATsinfo  gSfile, 7 ;duration
            seed      0

    instr PlayList
event_i "i", "PlayAll", 0, 1, 1, 0, .5 ;sine only, half speed
event_i "i", "PlayAll", giDur*2+1, 1, 0, 1, .5 ;noise only
event_i "i", "PlayAll", giDur*4+2, 1, .5, .5, .5 ;half sine, half noise
    endin

    instr PlayAll
iSinAmnt =      p4 ;sinee amount (0-1)
iNzAmnt  =      p5 ;noise amount (0-1)
iSpeed   =      p6 ;speed
p3       =      giDur/iSpeed
ktime    line   0, giDur/iSpeed, giDur
prints   "Resynthesizing all partials with tone = %.1f and noise = %.1f.\n", iSinAmnt, iNzAmnt
aOut     ATsinfo ktime, iSinAmnt, iNzAmnt, 1, gSfile, giNumParts
            outs   aOut, aOut
    endin

    instr PlayBand
iOffset  =      p4 ;offset in partials
iSpeed   =      p5 ;speed
p3       =      giDur/iSpeed
ktime    line   0, giDur/iSpeed, giDur
prints   "Resynthesizing partials %d to %d with related noise.\n", iOffset+1, iOffset+10
aOut     ATsinfo ktime, 1, 1, 1, gSfile, 10, iOffset, 1
            outs   aOut, aOut
;call itself again
if iOffset < giNumParts - 20 then
    event_i "i", "PlayBand", giDur/iSpeed+1, 1, iOffset+10, iSpeed
endif
    endin

    instr PlayWeighted
;sine amount, noise amount and speed are varying
kSinAmnt randomi 0, 1, 1, 3
kNzAmnt  =      1-kSinAmnt
kSpeed   randomi .01, .3, 1, 3
async    init    0
atime, aEnd syncphasor kSpeed/giDur, async
kTrig    metro   100
kEnd     max_k   aEnd, kTrig, 1 ;1 if phasor signal crosses zero
ktime    downsamp atime
aOut     ATsinfo ktime*giDur, kSinAmnt, kNzAmnt, 1, gSfile, giNumParts
            outs   aOut, aOut
;exit if file is at the end
if kEnd == 1 then
    event "i", "End", 0, 1
endif
    endin

    instr End
            exitnow
    endin

</CsInstruments>
<CsScore>
i "PlayList" 0 1
i "PlayBand" 20 1 0 .5
i "PlayWeighted" 110 100
</CsScore>
</CsSoundSynthesizer>

```

## See also

*ATSread, ATSreadnz, ATSinfo, ATSBufread, ATScross, ATSinterpread, ATSpartialtap, ATSadd, ATSaddnz*

## Credits

Author: Alex Norman  
Seattle, Washington  
2004

# babo

*babo* — A physical model reverberator.

## Description

*babo* stands for *ball-within-the-box*. It is a physical model reverberator based on the paper by Davide Rocchesso "The Ball within the Box: a sound-processing metaphor", *Computer Music Journal*, Vol 19, N.4, pp.45-47, Winter 1995.

The resonator geometry can be defined, along with some response characteristics, the position of the listener within the resonator, and the position of the sound source.

## Syntax

```
a1, a2 babo asig, ksrcx, ksrcy, ksrcz, irx, iry, irz [, idiff] [, ifno]
```

## Initialization

*irx*, *iry*, *irz* -- the coordinates of the geometry of the resonator (length of the edges in meters)

*idiff* -- is the coefficient of diffusion at the walls, which regulates the amount of diffusion (0-1, where 0 = no diffusion, 1 = maximum diffusion - default: 1)

*ifno* -- expert values function: a function number that holds all the additional parameters of the resonator. This is typically a GEN2--type function used in non-rescaling mode. They are as follows:

- *decay* -- main decay of the resonator (default: 0.99)
- *hydecay* -- high frequency decay of the resonator (default: 0.1)
- *rcvx*, *rcvy*, *rcvz* -- the coordinates of the position of the receiver (the listener) (in meters; 0,0,0 is the resonator center)
- *rdistance* -- the distance in meters between the two pickups (your ears, for example - default: 0.3)
- *direct* -- the attenuation of the direct signal (0-1, default: 0.5)
- *early\_diff* -- the attenuation coefficient of the early reflections (0-1, default: 0.8)

## Performance

*asig* -- the input signal

*ksrcx*, *ksrcy*, *ksrcz* -- the virtual coordinates of the source of sound (the input signal). These are allowed to move at k-rate and provide all the necessary variations in terms of response of the resonator.

## Examples

Here is a simple example of the *babo* opcode. It uses the file *babo.csd* [examples/babo.csd], and *beats.wav* [examples/beats.wav].

**Example 75. A simple example of the babo opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o babo.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

/* Written by Nicola Bernardini */

sr = 44100
ksmps = 32
nchnls = 2

; minimal babo instrument
;
instr 1
  ix      = p4 ; x position of source
  iy      = p5 ; y position of source
  iz      = p6 ; z position of source
  ixsize  = p7 ; width of the resonator
  iysize  = p8 ; depth of the resonator
  izsize  = p9 ; height of the resonator

  ainput soundin "beats.wav"

  al,ar babo   ainput*0.7, ix, iy, iz, ixsize, iysize, izsize

  outs    al,ar

endin

</CsInstruments>
<CsScore>

/* Written by Nicola Bernardini */
; simple babo usage:
;
;p4      : x position of source
;p5      : y position of source
;p6      : z position of source
;p7      : width of the resonator
;p8      : depth of the resonator
;p9      : height of the resonator
;
i 1 0 20 6 4 3 14.39 11.86 10
;          ^^^^^^  ^^^^^^^^^^^^^^^^^
;          |||/|||  ++++++::: optimal room dims according to
;          |||/|||  ++++++::: Milner and Bernard JASA 85(2), 1989
;          ++++++::: source position
e

</CsScore>
</CsoundSynthesizer>

```

Here is an advanced example of the babo opcode. It uses the file *babo\_expert.csd* [examples/babo\_expert.csd], and *beats.wav* [examples/beats.wav].

**Example 76. An advanced example of the babo opcode.**

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o babo_expert.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

/* Written by Nicola Bernardini */

sr = 44100
ksmps = 32
nchnls = 2

; full blown babo instrument with movement
;
instr 2
  ixstart = p4 ; start x position of source (left-right)
  ixend = p7 ; end x position of source
  iystart = p5 ; start y position of source (front-back)
  iyend = p8 ; end y position of source
  izstart = p6 ; start z position of source (up-down)
  izend = p9 ; end z position of source
  ixsize = p10 ; width of the resonator
  iysize = p11 ; depth of the resonator
  izsize = p12 ; height of the resonator
  idiff = p13 ; diffusion coefficient
  iexpert = p14 ; power user values stored in this function

  ainput soundin "beats.wav"
  ksource_x line ixstart, p3, ixend
  ksource_y line iystart, p3, iyend
  ksource_z line izstart, p3, izend

  al,ar babo ainput*0.7, ksource_x, ksource_y, ksource_z, ixsize, iysize, izsize, idiff, iexpert

  outs al,ar

endin

</CsInstruments>
<CsScore>

/* Written by Nicola Bernardini */
; full blown instrument
;p4 : start x position of source (left-right)
;p5 : end x position of source
;p6 : start y position of source (front-back)
;p7 : end y position of source
;p8 : start z position of source (up-down)
;p9 : end z position of source
;p10 : width of the resonator
;p11 : depth of the resonator
;p12 : height of the resonator
;p13 : diffusion coefficient
;p14 : power user values stored in this function

; decay hidecay rx ry rz rdistance direct early_diff
f1 0 8 -2 0.95 0.95 0 0 0 0.3 0.5 0.8 ; brighter
f2 0 8 -2 0.95 0.5 0 0 0 0.3 0.5 0.8 ; default (to be set as)
f3 0 8 -2 0.95 0.01 0 0 0 0.3 0.5 0.8 ; darker

```



# balance

balance — Adjust one audio signal according to the values of another.

## Description

The rms power of *asig* can be interrogated, set, or adjusted to match that of a comparator signal.

## Syntax

```
ares balance asig, acomp [, ihp] [, iskip]
```

## Initialization

*ihp* (optional) -- half-power point (in Hz) of a special internal low-pass filter. The default value is 10.

*iskip* (optional, default=0) -- initial disposition of internal data space (see *reson*). The default value is 0.

## Performance

*asig* -- input audio signal

*acomp* -- the comparator signal

*balance* outputs a version of *asig*, amplitude-modified so that its rms power is equal to that of a comparator signal *acomp*. Thus a signal that has suffered loss of power (eg., in passing through a filter bank) can be restored by matching it with, for instance, its own source. It should be noted that *gain* and *balance* provide amplitude modification only - output signals are not altered in any other respect.

## Examples

Here is an example of the *balance* opcode. It uses the file *balance.csd* [examples/balance.csd].

### Example 77. Example of the balance opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o balance.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
```

```
instr 1
; Generate a band-limited pulse train.
asrc buzz 0.9, 440, sr/440, 1

; Send the source signal through 2 filters.
a1 reson asrc, 1000, 100
a2 reson a1, 3000, 500

; Balance the filtered signal with the source.
afin balance a2, asrc
outs afin, afin

endin

</CsInstruments>
<CsScore>
; sine wave.
f 1 0 16384 10 1

i 1 0 2
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*gain, rms*

# bamboo

bamboo — Semi-physical model of a bamboo sound.

## Description

*bamboo* is a semi-physical model of a bamboo sound. It is one of the PhISEM percussion opcodes. PhISEM (Physically Informed Stochastic Event Modeling) is an algorithmic approach for simulating collisions of multiple independent sound producing objects.

## Syntax

```
ares bamboo kamp, idettack [, inum] [, idamp] [, imaxshake] [, ifreq] \  
    [, ifreq1] [, ifreq2]
```

## Initialization

*idettack* -- period of time over which all sound is stopped

*inum* (optional) -- The number of beads, teeth, bells, timbrels, etc. If zero, the default value is 1.25.

*idamp* (optional) -- the damping factor, as part of this equation:

$$\text{damping\_amount} = 0.9999 + (\text{idamp} * 0.002)$$

The default *damping\_amount* is 0.9999 which means that the default value of *idamp* is 0. The maximum *damping\_amount* is 1.0 (no damping). This means the maximum value for *idamp* is 0.05.

The recommended range for *idamp* is usually below 75% of the maximum value.

*imaxshake* (optional, default=0) -- amount of energy to add back into the system. The value should be in range 0 to 1.

*ifreq* (optional) -- the main resonant frequency. The default value is 2800.

*ifreq1* (optional) -- the first resonant frequency. The default value is 2240.

*ifreq2* (optional) -- the second resonant frequency. The default value is 3360.

## Performance

*kamp* -- Amplitude of output. Note: As these instruments are stochastic, this is only an approximation.

## Examples

Here is an example of the bamboo opcode. It uses the file *bamboo.csd* [examples/bamboo.csd].

### Example 78. Example of the bamboo opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o bamboo.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

instr 1
asig bamboo p4, 0.01,0, 0, 0, 8000
outs asig, asig

endin

</CsInstruments>
<CsScore>

i1 0 1 20000
i1 2 1 20000
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*dripwater, guiro, sleighbells, tambourine*

## Credits

Author: Perry Cook, part of the PhISEM (Physically Informed Stochastic Event Modeling)

Adapted by John ffitc

University of Bath, Codemist Ltd.

Bath, UK

New in Csound version 4.07

Added notes by Rasmus Ekman on May 2002.

# barmodel

barmodel — Creates a tone similar to a struck metal bar.

## Description

Audio output is a tone similar to a struck metal bar, using a physical model developed from solving the partial differential equation. There are controls over the boundary conditions as well as the bar characteristics.

## Syntax

```
ares barmodel kbcL, kbcR, iK, ib, kscan, iT30, ipos, ivel, iwid
```

## Initialization

*iK* -- dimensionless stiffness parameter. If this parameter is negative then the initialisation is skipped and the previous state of the bar is continued.

*ib* -- high-frequency loss parameter (keep this small).

*iT30* -- 30 db decay time in seconds.

*ipos* -- position along the bar that the strike occurs.

*ivel* -- normalized strike velocity.

*iwid* -- spatial width of strike.

## Performance

A note is played on a metallic bar, with the arguments as below.

*kbcL* -- Boundary condition at left end of bar (1 is clamped, 2 pivoting and 3 free).

*kbcR* -- Boundary condition at right end of bar (1 is clamped, 2 pivoting and 3 free).

*kscan* -- Speed of scanning the output location.

Note that changing the boundary conditions during playing may lead to glitches and is made available as an experiment. The use of a non-zero *kscan* can give apparent re-introduction of sound due to modulation.

## Examples

Here is an example of the barmodel opcode. It uses the file *barmodel.csd* [examples/barmodel.csd].

### Example 79. Example of the barmodel opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
```

```
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in   No messages
-odac         -iadc      -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o barmodel.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr           =          44100
kr           =          4410
ksmps       =           10
nchnls      =           1

; Instrument #1.
instr 1
  aq         barmodel  1, 1, p4, 0.001, 0.23, 5, p5, p6, p7
             out      aq
endin

</CsInstruments>
<CsScore>

i1 0.0 0.5  3 0.2 500  0.05
i1 0.5 0.5 -3 0.3 1000 0.05
i1 1.0 0.5 -3 0.4 1000 0.1
i1 1.5 4.0 -3 0.5 800  0.05
e
/* barmodel */

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Stefan Bilbao  
University of Edinburgh, UK  
Author: John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 5.01

# bbcutm

bbcutm — Generates breakbeat-style cut-ups of a mono audio stream.

## Description

The BreakBeat Cutter automatically generates cut-ups of a source audio stream in the style of drum and bass/jungle breakbeat manipulations. There are two versions, for mono (*bbcutm*) or stereo (*bbcuts*) sources. Whilst originally based on breakbeat cutting, the opcode can be applied to any type of source audio.

The prototypical cut sequence favoured over one bar with eighth note subdivisions would be

3+ 3R + 2

where we take a 3 unit block from the source's start, repeat it, then 2 units from the 7th and 8th eighth notes of the source.

We talk of rendering phrases (a sequence of cuts before reaching a new phrase at the beginning of a bar) and units (as subdivision th notes).

The opcode comes most alive when multiple synchronised versions are used simultaneously.

## Syntax

```
al bbcutm asource, ibps, isubdiv, ibarlength, iphrasebars, inumrepeats \  
    [, istutterspeed] [, istutterchance] [, ienvchoice ]
```

## Initialization

*ibps* -- Tempo to cut at, in beats per second.

*isubdiv* -- Subdivisions unit, for a bar. So 8 is eighth notes (of a 4/4 bar).

*ibarlength* -- How many beats per bar. Set to 4 for default 4/4 bar behaviour.

*iphrasebars* -- The output cuts are generated in phrases, each phrase is up to iphrasebars long

*inumrepeats* -- In normal use the algorithm would allow up to one additional repeat of a given cut at a time. This parameter allows that to be changed. Value 1 is normal- up to one extra repeat. 0 would avoid repeating, and you would always get back the original source except for enveloping and stuttering.

*istutterspeed* -- (optional, default=1) The stutter can be an integer multiple of the subdivision speed. For instance, if subdiv is 8 (quavers) and stutterspeed is 2, then the stutter is in semiquavers (sixteenth notes= subdiv 16). The default is 1.

*istutterchance* -- (optional, default=0) The tail of a phrase has this chance of becoming a single repeating one unit cell stutter (0.0 to 1.0). The default is 0.

*ienvchoice* -- (optional, default=1) choose 1 for on (exponential envelope for cut grains) or 0 for off. Off will cause clicking, but may give good noisy results, especially for percussive sources. The default is 1, on.

## Performance

*asource* -- The audio signal to be cut up. This version runs in real-time without knowledge of future audio.

## Examples

Here is a simple example of the *bbcutm* opcode. It uses the file *bbcutm.csd* [examples/bbcutm.csd], and *beats.wav* [examples/beats.wav].

### Example 80. A simple example of the *bbcutm* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o bbcutm.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - Play an audio file normally.
instr 1
  asource soundin "beats.wav"
  out asource
endin

; Instrument #2 - Cut-up an audio file.
instr 2
  asource soundin "beats.wav"

  ibps = 4
  isubdiv = 8
  ibarlength = 4
  iphrasebars = 1
  inumrepeats = 2

  a1 bbcutm asource, ibps, isubdiv, ibarlength, iphrasebars, inumrepeats

  out a1
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for two seconds.
i 1 0 2
; Play Instrument #2 for two seconds.
i 2 3 2
e

```

```
</CsScore>
</CsoundSynthesizer>
```

Here are some more advanced examples...

### Example 81. First steps- mono and stereo versions

```
<CsoundSynthesizer>
<CsInstruments>
sr      =      44100
kr      =      4410
ksmps  =      10
nchnls =      2

instr 1
  asource diskin "break7.wav",1,0,1 ; a source breakbeat sample, wraparound lest it stop!

  ; cuts in eighth notes per 4/4 bar, up to 4 bar phrases, up to 1
  ; repeat in total (standard use) rare stuttering at 16 note speed,
  ; no enveloping
  asig bbcutm asource, 2.6937, 8, 4, 4, 1, 2, 0.1, 0

  outs      asig,asig
endin

instr 2 ;stereo version
  asource1,asource2 diskin "break7stereo.wav", 1, 0, 1 ; a source breakbeat sample, wraparound lest

  ; cuts in eighth notes per 4/4 bar, up to 4 bar phrases, up to 1
  ; repeat in total (standard use) rare stuttering at 16 note speed,
  ; no enveloping
  asigl,asig2 bbcuts asource1, asource2, 2.6937, 8, 4, 4, 1, 2, 0.1, 0

  outs asigl,asig2
endin

</CsInstruments>
<CsScore>
i1 0 10
i2 11 10
e
</CsScore>
</CsoundSynthesizer>
```

### Example 82. Multiple simultaneous synchronised breaks

```
<CsoundSynthesizer>
<CsInstruments>
sr      =      44100
kr      =      4410
ksmps  =      10
nchnls =      2

instr 1
  ibps = 2.6937
  iplaybackspeed = ibps/p5
  asource diskin p4, iplaybackspeed, 0, 1

  asig bbcutm asource, 2.6937, p6, 4, 4, p7, 2, 0.1, 1

  out asig
endin

</CsInstruments>
```

```

<CsScore>

; source      bps cut repeats
i1 0 10 "break1.wav" 2.3 8 2 //2.3 is the source original tempo
i1 0 10 "break2.wav" 2.4 8 3
i1 0 10 "break3.wav" 2.5 16 4
e
</CsScore>
</CsoundSynthesizer>

```

**Example 83. Cutting up any old audio- much more interesting noises than this should be possible!**

```

<CsoundSynthesizer>
<CsInstruments>
sr      =      44100
kr      =      4410
ksmps  =      10
nchnls =      2

instr 1
  asource oscil 20000, 70, 1
  ; ain, bps, subdiv, barlength, phrasebars, numrepeats,
  ; stutterspeed, stutterchance, envelopingon
  asig bbcutm asource, 2, 32, 1, 1, 2, 4, 0.6, 1
  outs asig
endin

</CsInstruments>
<CsScore>
f1 0 256 10 1
i1 0 10
e
</CsScore>
</CsoundSynthesizer>

```

**Example 84. Constant stuttering- faked, not possible since can only stutter in last half bar could make extra stuttering option parameter**

```

<CsoundSynthesizer>
<CsInstruments>
sr      =      44100
kr      =      4410
ksmps  =      10
nchnls =      2

instr 1
  asource disk "break7.wav", 1, 0, 1

  ;16th note cuts- but cut size 2 over half a beat.
  ;each half beat will either survive intact or be turned into
  ;the first sixteenth played twice in succession

  asig bbcutm asource, 2.6937, 2, 0.5, 1, 2, 2, 1.0, 0
  outs asig
endin

</CsInstruments>
<CsScore>
i1 0 30
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*bbcuts*

## Credits

Author: Nick Collins  
London  
August 2001

New in version 4.13

# bbcuts

bbcuts — Generates breakbeat-style cut-ups of a stereo audio stream.

## Description

The BreakBeat Cutter automatically generates cut-ups of a source audio stream in the style of drum and bass/jungle breakbeat manipulations. There are two versions, for mono (*bbcutm*) or stereo (*bbcuts*) sources. Whilst originally based on breakbeat cutting, the opcode can be applied to any type of source audio.

The prototypical cut sequence favoured over one bar with eighth note subdivisions would be

3+ 3R + 2

where we take a 3 unit block from the source's start, repeat it, then 2 units from the 7th and 8th eighth notes of the source.

We talk of rendering phrases (a sequence of cuts before reaching a new phrase at the beginning of a bar) and units (as subdivision th notes).

The opcode comes most alive when multiple synchronised versions are used simultaneously.

## Syntax

```
a1,a2 bbcuts asource1, asource2, ibps, isubdiv, ibarlength, iphrasebars, \  
inumrepeats [, istutterspeed] [, istutterchance] [, ienvchoice]
```

## Initialization

*ibps* -- Tempo to cut at, in beats per second.

*isubdiv* -- Subdivisions unit, for a bar. So 8 is eighth notes (of a 4/4 bar).

*ibarlength* -- How many beats per bar. Set to 4 for default 4/4 bar behaviour.

*iphrasebars* -- The output cuts are generated in phrases, each phrase is up to iphrasebars long

*inumrepeats* -- In normal use the algorithm would allow up to one additional repeat of a given cut at a time. This parameter allows that to be changed. Value 1 is normal- up to one extra repeat. 0 would avoid repeating, and you would always get back the original source except for enveloping and stuttering.

*istutterspeed* -- (optional, default=1) The stutter can be an integer multiple of the subdivision speed. For instance, if subdiv is 8 (quavers) and stutterspeed is 2, then the stutter is in semiquavers (sixteenth notes= subdiv 16). The default is 1.

*istutterchance* -- (optional, default=0) The tail of a phrase has this chance of becoming a single repeating one unit cell stutter (0.0 to 1.0). The default is 0.

*ienvchoice* -- (optional, default=1) choose 1 for on (exponential envelope for cut grains) or 0 for off. Off will cause clicking, but may give good noisy results, especially for percussive sources. The default is 1, on.

## Performance

*asource* -- The audio signal to be cut up. This version runs in real-time without knowledge of future audio.

## Examples

Here is an example of the *bbcuts* opcode. It uses the file *bbcuts.csd* [examples/bbcuts.csd].

### Example 85. Example of the *bbcuts* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o bbcuts.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

instr 1 ;Play an audio file
aleft, aright diskin2 "kickroll.wav", 1, 0
outs aleft, aright

endin

instr 2 ;Cut-up stereo audio file.

ibps = 16
isubdiv = 2
ibarlength = 2
iphasebars = 1
inumrepeats = 8

aleft, aright diskin2 "kickroll.wav", 1, 0
aleft, aright bbcuts aleft, aright, ibps, isubdiv, ibarlength, iphasebars, inumrepeats
outs aleft, aright

endin

</CsInstruments>
<CsScore>

i 1 0 2
i 2 3 2
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*bbcutm*

## Credits

Author: Nick Collins  
London  
August 2001

New in version 4.13

# betarand

betarand — Beta distribution random number generator (positive values only).

## Description

Beta distribution random number generator (positive values only). This is an x-class noise generator.

## Syntax

```
ares betarand krange, kalpha, kbeta
ires betarand krange, kalpha, kbeta
kres betarand krange, kalpha, kbeta
```

## Performance

*krange* -- range of the random numbers (0 - *krange*).

*kalpha* -- alpha value. If *kalpha* is smaller than one, smaller values favor values near 0.

*kbeta* -- beta value. If *kbeta* is smaller than one, smaller values favor values near *krange*.

If both *kalpha* and *kbeta* equal one we have uniform distribution. If both *kalpha* and *kbeta* are greater than one we have a sort of Gaussian distribution. Outputs only positive numbers.

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the betarand opcode. It uses the file *betarand.csd* [examples/betarand.csd].

### Example 86. Example of the betarand opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o betarand.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; every run time same values

kbeta betarand 100, 1, 1
  printk .2, kbeta ; look
aout oscili 0.8, 440+kbeta, 1 ; & listen
  outs aout, aout
endin

instr 2 ; every run time different values

  seed 0
kbeta betarand 100, 1, 1
  printk .2, kbeta ; look
aout oscili 0.8, 440+kbeta, 1 ; & listen
  outs aout, aout
endin

</CsInstruments>
<CsScore>
; sine wave
f 1 0 16384 10 1

i 1 0 2
i 2 3 2
e

</CsScore>
</CsoundSynthesizer>

```

Its output should include a line like:

```

i 1 1 time      0.00267:    85.74227
i 1 1 time      0.20267:    12.07606
i 1 1 time      0.40267:    25.03239
i 1 1 time      0.60267:     0.42037
i 1 1 time      0.80267:    76.69589
i 1 1 time      1.00000:    29.73339
i 1 1 time      1.20267:    48.29811
i 1 1 time      1.40267:    75.46507
i 1 1 time      1.60267:    74.80686
i 1 1 time      1.80000:    81.37473
i 1 1 time      2.00000:    55.48827
Seeding from current time 3472120656
i 2 2 time      3.00267:    57.21408
i 2 2 time      3.20267:    30.95705
i 2 2 time      3.40267:    19.71687
i 2 2 time      3.60000:    64.48965
i 2 2 time      3.80267:    72.35818
i 2 2 time      4.00000:    49.65395
i 2 2 time      4.20000:    55.25888
i 2 2 time      4.40267:     3.98308
i 2 2 time      4.60267:    52.98075
i 2 2 time      4.80267:    58.07925
i 2 2 time      5.00000:    56.38914

```

## See Also

*seed, bexprnd, cauchy, exprand, gauss, linrand, pcauchy, poisson, trirand, unirand, weibull*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

Existed in 3.30

# bexprnd

bexprnd — Exponential distribution random number generator.

## Description

Exponential distribution random number generator. This is an x-class noise generator.

## Syntax

```
ares bexprnd krange
ires bexprnd krange
kres bexprnd krange
```

## Performance

*krange* -- the range of the random numbers (*-krange* to *+krange*)

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the bexprnd opcode. It uses the file *bexprnd.csd* [examples/bexprnd.csd].

### Example 87. Example of the bexprnd opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o bexprnd.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; every run time same values

kexp bexprnd 100
printk .2, kexp ; look
```

```

aout oscili 0.8, 440+kexp, 1 ; & listen
outs aout, aout

endin

instr 2 ; every run time different values

seed 0
kexp bexprnd 100
printk .2, kexp ; look
aout oscili 0.8, 440+kexp, 1 ; & listen
outs aout, aout
endin

</CsInstruments>
<CsScore>
; sine wave
f 1 0 16384 10 1

i 1 0 2
i 2 3 2
e

</CsScore>
</CsoundSynthesizer>

```

Its output should include a line like:

```

i 1 time 0.00267: 99.27598
i 1 time 0.20267: 74.97176
i 1 time 0.40267: -35.67213
i 1 time 0.60267: 1.10579
i 1 time 0.80267: -18.08816
i 1 time 1.00000: 28.93329
i 1 time 1.20267: 320.63733
i 1 time 1.40267: -332.05614
i 1 time 1.60267: -212.66361
i 1 time 1.80000: -92.57433
i 1 time 2.00000: 140.70939
Seeding from current time 4055201702
i 2 time 3.00267: 190.30495
i 2 time 3.20267: -58.30677
i 2 time 3.40267: 192.39784
i 2 time 3.60000: 12.72448
i 2 time 3.80267: 79.91503
i 2 time 4.00000: 34.44258
i 2 time 4.20000: 167.92680
i 2 time 4.40267: -117.10278
i 2 time 4.60267: -70.99155
i 2 time 4.80267: -23.24037
i 2 time 5.00000: -226.35500

```

## See Also

*seed, betarand, cauchy, exprand, gauss, linrand, pcauchy, poisson, trirand, unirand, weibull*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

# bformenc1

bformenc1 — Codes a signal into the ambisonic B format.

## Description

Codes a signal into the ambisonic B format

## Syntax

```
aw, ax, ay, az bformenc1 asig, kalpha, kbeta
aw, ax, ay, az, ar, as, at, au, av bformenc1 asig, kalpha, kbeta
aw, ax, ay, az, ar, as, at, au, av, ak, al, am, an, ao, ap, aq bformenc1 \
    asig, kalpha, kbeta
aarray[] bformenc1 asig, kalpha, kbeta
```

## Performance

*aw, ax, ay, ...* -- output cells of the B format.

*aarray* -- output array to hold cells of the B format.

*asig* -- input signal.

*kalpha* -- azimuth angle in degrees (anticlockwise).

*kbeta* -- altitude angle in degrees.

## Example

Here is an example of the bformenc1 opcode. It uses the file *bformenc1.csd* [examples/bformenc1.csd].

### Example 88. Example of the bformenc1 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
;-odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
-o bformenc.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 8
0dbfs = 1

instr 1 ;without arrays
; generate pink noise
```

```
anoise pinkish 1

; two full turns
kalpha line 0, p3, 720
kbeta = 0

; generate B format
aw, ax, ay, az, ar, as, at, au, av bformenc1 anoise, kalpha, kbeta

; decode B format for 8 channel circle loudspeaker setup
a1, a2, a3, a4, a5, a6, a7, a8 bformdec1 4, aw, ax, ay, az, ar, as, at, au, av

; write audio out
outo a1, a2, a3, a4, a5, a6, a7, a8
endin

instr 2 ;with arrays (csound6)
;set file names for:
S_bform = "bform_out.wav" ;b-format (2nd order) output
S_sound = "sound_out.wav" ;sound output

; generate pink noise
anoise pinkish 1

; two full turns
kalpha line 0, p3, 720
kbeta = 0

;create array for B-format 2nd order (9 chnls)
aBform[] init 9

; generate B-format
aBform bformenc1 anoise, kalpha, kbeta

;write out b-format
fout "fout.wav", 18, aBform

;create array for audio output (8 channels)
aAudio[] init 8

;decode B format for 8 channel circle loudspeaker setup
aAudio bformdec1 4, aBform

; write audio out
fout S_sound, 18, aAudio
endin

</CsInstruments>
<CsScore>
i 1 0 8
i 2 8 8
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*bformdec1*

## Credits

Author: Richard Furse, Bruce Wiggins and Fons Adriaensen, following code by Samuel Groner  
2008

New in version 5.09

Array output new in version 6.01

# bformdec1

bformdec1 — Decodes an ambisonic B format signal

## Description

Decodes an ambisonic B format signal into loudspeaker specific signals.

## Syntax

```

ao1, ao2 bformdec1 isetup, aw, ax, ay, az [, ar, as, at, au, av \
    [, abk, al, am, an, ao, ap, aq]]

ao1, ao2, ao3, ao4 bformdec1 isetup, aw, ax, ay, az [, ar, as, at, \
    au, av [, abk, al, am, an, ao, ap, aq]]

ao1, ao2, ao3, ao4, ao5 bformdec1 isetup, aw, ax, ay, az [, ar, as, \
    at, au, av [, abk, al, am, an, ao, ap, aq]]

ao1, ao2, ao3, ao4, ao5, ao6, ao7, ao8 bformdec1 isetup, aw, ax, ay, az \
    [, ar, as, at, au, av [, abk, al, am, an, ao, ap,
    aq]]

aout[] bformdec1 isetup, abform[]

```

## Initialization

Note that horizontal angles are measured anticlockwise in this description.

*isetup* — loudspeaker setup. There are five supported setups:

- 1. Stereo - L(90), R(-90); this is an M+S style stereo decode.
- 2. Quad - FL(45), BL(135), BR(-135), FR(-45). This is a first-order `in-phase' decode.
- 3. 5.0 - L(30),R(-30),C(0),BL(110),BR(-110). Note that many people do not actually use the angles above for their speaker arrays and a good decode for DVD etc can be achieved using the Quad configuration to feed L, R, BL and BR (leaving C silent).
- 4. Octagon - FFL(22.5),FLL(67.5),BLL(112.5),BBL(157.5),BBR(-157.5),BRR(-112.5),FRR(-67.5),F-  
FR(-22.5). This is a first-, second- or third-order `in-phase' decode, depending on the number of input channels.
- 5. Cube  
FLD(45,-35.26),FLU(45,35.26),BLD(135,-35.26),BLU(135,35.26),BRD(-135,-35.26),BRU(-135,35.26),FRD(-45,-35.26)  
This is a first-order `in-phase' decode.

## Performance

*aw, ax, ay, ...* -- input signal in the B format.

*ao1 .. ao8* — loudspeaker specific output signals.

## Example

Here is an example of the `bformdec1` opcode. It uses the file `bformenc1.csd` [examples/bformenc1.csd].

### Example 89. Example of the `bformdec1` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
;-odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
-o bformenc.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 8
Odbfs = 1

instr 1 ;without arrays
; generate pink noise
anoise pinkish 1

; two full turns
kalpha line 0, p3, 720
kbeta = 0

; generate B format
aw, ax, ay, az, ar, as, at, au, av bformenc1 anoise, kalpha, kbeta

; decode B format for 8 channel circle loudspeaker setup
a1, a2, a3, a4, a5, a6, a7, a8 bformdec1 4, aw, ax, ay, az, ar, as, at, au, av

; write audio out
outo a1, a2, a3, a4, a5, a6, a7, a8
endin

instr 2 ;with arrays (csound6)
;set file names for:
S_bform = "bform_out.wav" ;b-format (2nd order) output
S_sound = "sound_out.wav" ;sound output

; generate pink noise
anoise pinkish 1

; two full turns
kalpha line 0, p3, 720
kbeta = 0

;create array for B-format 2nd order (9 chnls)
aBform[] init 9

; generate B-format
aBform bformenc1 anoise, kalpha, kbeta

;write out b-format
fout "fout.wav", 18, aBform

;create array for audio output (8 channels)
aAudio[] init 8

```

```
;decode B format for 8 channel circle loudspeaker setup
aAudio bformdecl 4, aBform

; write audio out
fout S_sound, 18, aAudio
endin

</CsInstruments>
<CsScore>
i 1 0 8
i 2 8 8
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*bformenc1*

## Credits

Author: Richard Furse, Bruce Wiggins and Fons Adriaensen, following code by Samuel Groner 2008

New in version 5.09

# binit

binit — PVS tracks to amplitude+frequency conversion.

## Description

The binit opcode takes an input containing a TRACKS pv streaming signal (as generated, for instance by partials) and converts it into an equal-bandwidth bin-frame containing amplitude and frequency pairs (PVS\_AMP\_FREQ), suitable for overlap-add resynthesis (such as performed by pvsynth) or further PVS streaming phase vocoder signal transformations. For each frequency bin, it will look for a suitable track signal to fill it; if not found, the bin will be empty (0 amplitude). If more than one track fits a certain bin, the one with highest amplitude will be chosen. This means that not all of the input signal is actually 'binned', the operation is lossy. However, in many situations this loss is not perceptually relevant.

## Syntax

```
fsig binit fin, isize
```

## Performance

*fsig* -- output pv stream in PVS\_AMP\_FREQ format

*fin* -- input pv stream in TRACKS format

*isize* -- FFT size of output (N).

## Examples

Here is an example of the binit opcode. It uses the file *binit.csd* [examples/binit.csd].

### Example 90. Example of the binit opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o binit.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

instr 1
;ain inch 1      ; for live input
ain diskin "beats.wav", 1 ; input signal
fs1,fsi2 pvsifd ain, 2048, 512, 1 ; ifd analysis
```

```
fst partials fs1, fsi2, .003, 1, 3, 500 ; partial tracking
fbins binit fst, 2048 ; convert it back to bins
aout pvsynth fbins ; overlap-add resynthesis
outs aout, aout

endin

</CsInstruments>
<CsScore>

i 1 0 2
e

</CsScore>
</CsoundSynthesizer>
```

The example above shows partial tracking of an ifd-analysis signal, conversion to bin frames and overlap-add resynthesis.

## Credits

Author: Victor Lazzarini  
February 2006

New in Csound5.01

# biquad

biquad — A sweepable general purpose biquadratic digital filter.

## Description

A sweepable general purpose biquadratic digital filter.

## Syntax

```
ares biquad asig, kb0, kb1, kb2, ka0, ka1, ka2 [, iskip]
```

## Initialization

*iskip* (optional, default=0) -- if non-zero, initialization will be skipped. Default value 0. (New in Csound version 3.50)

## Performance

*asig* -- input signal

*biquad* is a general purpose biquadratic digital filter of the form:

$$a0*y(n) + a1*y[n-1] + a2*y[n-2] = b0*x[n] + b1*x[n-1] + b2*x[n-2]$$

This filter has the following frequency response:

$$H(Z) = \frac{B(Z)}{A(Z)} = \frac{b0 + b1*Z^{-1} + b2*Z^{-2}}{a0 + a1*Z^{-1} + a2*Z^{-2}}$$

This type of filter is often encountered in digital signal processing literature. It allows six user-defined k-rate coefficients.

## Examples

Here is an example of the biquad opcode. It uses the file *biquad.csd* [examples/biquad.csd].

### Example 91. Example of the biquad opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o biquad.wav -W ;; for file output any platform
</CsOptions>
```

```

<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

; Instrument #1.
instr 1
; Get the values from the score.
idur = p3
iamp = p4
icps = cpspch(p5)
kfco = p6
krez = p7

; Calculate the biquadratic filter's coefficients
kfcon = 2*3.14159265*kfco/sr
kalpha = 1-2*krez*cos(kfcon)*cos(kfcon)+krez*krez*cos(2*kfcon)
kbeta = krez*krez*sin(2*kfcon)-2*krez*cos(kfcon)*sin(kfcon)
kgama = 1+cos(kfcon)
km1 = kalpha*kgama+kbeta*sin(kfcon)
km2 = kalpha*kgama-kbeta*sin(kfcon)
kden = sqrt(km1*km1+km2*km2)
kb0 = 1.5*(kalpha*kalpha+kbeta*kbeta)/kden
kb1 = kb0
kb2 = 0
ka0 = 1
ka1 = -2*krez*cos(kfcon)
ka2 = krez*krez

; Generate an input signal.
axn vco 1, icps, 1

; Filter the input signal.
ayn biquad axn, kb0, kb1, kb2, ka0, ka1, ka2
outs ayn*iamp/2, ayn*iamp/2
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Sta Dur Amp Pitch Fco Rez
i 1 0.0 1.0 20000 6.00 1000 .8
i 1 1.0 1.0 20000 6.03 2000 .95
e

</CsScore>
</CsSoundSynthesizer>

```

Here is another example of the biquad opcode used for modal synthesis. It uses the file *biquad-2.csd* [examples/biquad-2.csd]. See the *Modal Frequency Ratios* appendix for other frequency ratios.

### Example 92. Example of the biquad opcode for modal synthesis.

```

<CsSoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out Audio in

```

```

-odac          -iadc    ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o biquad-2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
sr = 44100
kr = 4410
kmps = 10
nchnls = 2

/* modal synthesis using biquad filters as oscillators
   Example by Scott Lindroth 2007 */

instr 1

  ipi = 3.1415926
  idenom = sr*0.5

  ipulseSpd = p4
  icps      = p5
  ipan      = p6
  iamp      = p7
  iModes    = p8

  apulse    mpulse iamp, 0

  icps      = cpspch( icps )

  ; filter gain

  iamp1 = 600
  iamp2 = 1000
  iamp3 = 1000
  iamp4 = 1000
  iamp5 = 1000
  iamp6 = 1000

  ; resonance

  irpole1 = 0.99999
  irpole2 = irpole1
  irpole3 = irpole1
  irpole4 = irpole1
  irpole5 = irpole1
  irpole6 = irpole1

  ; modal frequencies

  if (iModes == 1) goto modes1
  if (iModes == 2) goto modes2

modes1:
  if1    = icps * 1           ;pot lid
  if2    = icps * 6.27
  if3    = icps * 3.2
  if4    = icps * 9.92
  if5    = icps * 14.15
  if6    = icps * 6.23
  goto nextPart

modes2:
  if1    = icps * 1           ;uniform wood bar
  if2    = icps * 2.572
  if3    = icps * 4.644
  if4    = icps * 6.984
  if5    = icps * 9.723

```

```

if6      = icps * 12.0
goto nextPart

nextPart:

; convert frequency to radian frequency

itheta1 = (if1/idenom) * ipi
itheta2 = (if2/idenom) * ipi
itheta3 = (if3/idenom) * ipi
itheta4 = (if4/idenom) * ipi
itheta5 = (if5/idenom) * ipi
itheta6 = (if6/idenom) * ipi

; calculate coefficients

ib11 = -2 * irpole1 * cos(itheta1)
ib21 = irpole1 * irpole1
ib12 = -2 * irpole2 * cos(itheta2)
ib22 = irpole2 * irpole2
ib13 = -2 * irpole3 * cos(itheta3)
ib23 = irpole3 * irpole3
ib14 = -2 * irpole4 * cos(itheta4)
ib24 = irpole4 * irpole4
ib15 = -2 * irpole5 * cos(itheta5)
ib25 = irpole5 * irpole5
ib16 = -2 * irpole6 * cos(itheta6)
ib26 = irpole6 * irpole6

;printk 1, ib 11
;printk 1, ib 21

; also try setting the -1 coeff. to 0, but be sure to scale down the amplitude!

asin1    biquad  apulse * iamp1, 1, 0, -1, 1, ib11, ib21
asin2    biquad  apulse * iamp2, 1, 0, -1, 1, ib12, ib22
asin3    biquad  apulse * iamp3, 1, 0, -1, 1, ib13, ib23
asin4    biquad  apulse * iamp4, 1, 0, -1, 1, ib14, ib24
asin5    biquad  apulse * iamp5, 1, 0, -1, 1, ib15, ib25
asin6    biquad  apulse * iamp6, 1, 0, -1, 1, ib16, ib26

afin     =      (asin1 + asin2 + asin3 + asin4 + asin5 + asin6)

outs     afin * sqrt(p6), afin*sqrt(1-p6)

endin
</CsInstruments>
<CsScore>
;ins    st    dur  pulseSpd  pch    pan    amp    Modes
i1      0    12   0          7.089  0      0.7    2
i1      .    .    .          7.09   1      .      .
i1      .    .    .          7.091  0.5    .      .

i1      0    12   0          8.039  0      0.7    2
i1      0    12   0          8.04   1      0.7    2
i1      0    12   0          8.041  0.5    0.7    2

i1      9    .    .          7.089  0      .      2
i1      .    .    .          7.09   1      .      .
i1      .    .    .          7.091  0.5    .      .

i1      9    12   0          8.019  0      0.7    2
i1      9    12   0          8.02   1      0.7    2
i1      9    12   0          8.021  0.5    0.7    2
e
</CsScore>

```

`</CsoundSynthesizer>`

## See Also

*biquada, moogvcf, rezzv*

## Credits

Author: Hans Mikelson  
October 1998

New in Csound version 3.49

# biquada

`biquada` — A sweepable general purpose biquadratic digital filter with a-rate parameters.

## Description

A sweepable general purpose biquadratic digital filter.

## Syntax

```
ares biquada asig, ab0, ab1, ab2, aa0, aa1, aa2 [, iskip]
```

## Initialization

`iskip` (optional, default=0) -- if non-zero, initialization will be skipped. Default value 0. (New in Csound version 3.50)

## Performance

`asig` -- input signal

`biquada` is a general purpose biquadratic digital filter of the form:

$$a0*y(n) + a1*y[n-1] + a2*y[n-2] = b0*x[n] + b1*x[n-1] + b2*x[n-2]$$

This filter has the following frequency response:

$$H(Z) = \frac{B(Z)}{A(Z)} = \frac{b0 + b1*Z^{-1} + b2*Z^{-2}}{a0 + a1*Z^{-1} + a2*Z^{-2}}$$

This type of filter is often encountered in digital signal processing literature. It allows six user-defined a-rate coefficients.

## Examples

Here is an example of the `biquada` opcode. It uses the file `biquad.csd` [examples/biquada.csd].

### Example 93. Example of the `biquada` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc      -d          ;;RT audio I/O
```

```

; For Non-realtime ouput leave only the line below:
; -o biquad.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

; Instrument #1.
instr 1
; Get the values from the score.
idur = p3
iamp = p4
icps = cpspch(p5)
afco expon 100, p3, 2000
arez line 0.8, p3, 0.99

; Calculate the biquadric filter's coefficients
afcon = 2*3.14159265*afco/sr
aalpha = 1-2*arez*cos(afcon)*cos(afcon)+arez*arez*cos(2*afcon)
abeta = arez*arez*sin(2*afcon)-2*arez*cos(afcon)*sin(afcon)
agama = 1+cos(afcon)
am1 = aalpha*agama+abeta*sin(afcon)
am2 = aalpha*agama-abeta*sin(afcon)
aden = sqrt(am1*am1+am2*am2)
ab0 = 1.5*(aalpha*aalpha+abeta*abeta)/aden
ab1 = ab0
ab2 = 0
aa0 = 1
aa1 = -2*arez*cos(afcon)
aa2 = arez*arez

; Generate an input signal.
axn vco 1, icps, 1

; Filter the input signal.
ayn biquada axn, ab0, ab1, ab2, aa0, aa1, aa2
outs ayn*iamp/2, ayn*iamp/2
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Sta Dur Amp Pitch
i 1 0.0 5.0 20000 6.00
e

</CsScore>
</CsoundSynthesizer>

```

## See Also

*biquad*

## Credits

Author: John ffitch after Hans Mikelson

August 2001

New in Csound version 4.13

# birnd

birnd — Returns a random number in a bi-polar range.

## Description

Returns a random number in a bi-polar range.

## Syntax

`birnd(x)` (init- or control-rate only)

Where the argument within the parentheses may be an expression. These value converters sample a global random sequence, but do not reference *seed*. The result can be a term in a further expression.

## Performance

Returns a random number in the bipolar range  $-x$  to  $x$ . *rnd* and *birnd* obtain values from a global pseudo-random number generator, then scale them into the requested range. The single global generator will thus distribute its sequence to these units throughout the performance, in whatever order the requests arrive.

## Examples

Here is an example of the birnd opcode. It uses the file *birnd.csd* [examples/birnd.csd].

### Example 94. Example of the birnd opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

instr 1 ; Generate a random number from -1 to 1.

kbin = birnd(1)
printk .2, kbin

endin

</CsInstruments>
<CsScore>

i 1 0 1
i 1 + .
```

```
i 1 + .  
i 1 + .  
i 1 + .  
i 1 + .  
i 1 + .
```

```
e
```

```
</CsScore>  
</CsoundSynthesizer>
```

Its output should include lines like:

```
instr 1: i1 = 0.94700  
instr 1: i1 = -0.72119  
instr 1: i1 = 0.53898  
instr 1: i1 = 0.05001  
instr 1: i1 = 0.24533  
instr 1: i1 = 0.93902  
instr 1: i1 = 0.43364
```

## See Also

*rnd*

## Credits

Author: Barry L. Vercoe  
MIT  
Cambridge, Massachusetts  
1997

Extended in 3.47 to x-rate by John fitch.

# bpf

bpf — Break point function with linear interpolation

## Description

Break-point function with linear interpolation. Useful when defining a table with GEN27 and scaling the x value would be overkill.

## Syntax

```
ky bpf kx, kx1, ky1, kx2, ..., kxn, kyn
```

## Performance

**kx** -- Input value

**kxn, kyn** -- Defines a breakpoint. Can be changed at `krate`, but all *kx*s must be sorted

The points  $(kx_1, ky_1)$ ,  $(kx_2, ky_2)$ , etc, define a linearly interpolated function. This function is evaluated at point *kx*. This function extends to both `-inf` and `+inf`, so if  $kx < kx_1$  then  $ky = ky_1$  and the same holds true the other end.

These are equivalent:

```
ky bpf kx, 0, 0, 0.5, 10, 1.02, 200

itab ftgenonce 0, 0, -27, 0, 0, 50, 10, 102, 200
ky = tablei(limit(kx, 0, 1.02)*100, itab)
```



### Note

x values need to be sorted. Both x and y values can change but x values must stay sorted.

## Examples

Here is an example of the `bpf` opcode. It uses the file `bpf.csd` [examples/bpf.csd].

### Example 95. Example of the `bpf` opcode.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>

ksmps = 128
nchnls = 2

instr 1
  kx line -1, p3, 2.5
```

```
ky bpf kx,      \
    0,    0,    \
    1.01, 10,   \
    2,    0.5,  \
    2.5,  -1

; the same expressed as a table (notice that we need to scale and
; limit kx). It gets even more complicated with breakpoints with
; negative x
itab ftgenonce 0, 0, 1000, -27, \
    0, 0, \
    101, 10, \
    200, 0.5, \
    250, -1, \
    1000, -1

ky2 tablei limit(kx, 0, 2.5)*100, itab

printks "kx: %f   ky: %f   ky2: %f \n", 1/kr, kx, ky, ky2

endin

</CsInstruments>
<CsScore>
i 1 0 2

</CsScore>
</CsoundSynthesizer>
```

## See Also

*scale, ntrpol, xyscale, linlin*

## Credits

By: Eduardo Moguillansky 2017

# bqrez

bqrez — A second-order multi-mode filter.

## Description

A second-order multi-mode filter.

## Syntax

```
ares bqrez asig, xfco, xres [, imode] [, iskip]
```

## Initialization

*imode* (optional, default=0) -- The mode of the filter. Choose from one of the following:

- 0 = low-pass (default)
- 1 = high-pass
- 2 = band-pass
- 3 = band-reject
- 4 = all-pass

*iskip* (optional, default=0) -- if non zero skip the initialisation of the filter. (New in Csound version 4.23f13 and 5.0)

## Performance

*ares* -- output audio signal.

*asig* -- input audio signal.

*xfco* -- filter cut-off frequency in Hz. May be i-time, k-rate, a-rate.

*xres* -- amount of resonance. Values of 1 to 100 are typical. Resonance should be one or greater. A value of 100 gives a 20dB gain at the cutoff frequency. May be i-time, k-rate, a-rate.

All filter modes can be frequency modulated as well as the resonance can also be frequency modulated.

*bqrez* is a resonant low-pass filter created using the Laplace s-domain equations for low-pass, high-pass, and band-pass filters normalized to a frequency. The bi-linear transform was used which contains a frequency transform constant from s-domain to z-domain to exactly match the frequencies together. A lot of trigonometric identities were used to simplify the calculation. It is very stable across the working frequency range up to the Nyquist frequency.

## Examples

Here is an example of the *bqrez* opcode. It uses the file *bqrez.csd* [examples/bqrez.csd].

## Example 96. Example of the `bqrez` opcode borrowed from the “`rezzy`” opcode in Kevin Conder's manual.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o bqrez.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1    ;sawtooth waveform.

kfco line 200, p3, 2000;filter-cutoff frequency from .2 to 5 KHz.
kres = p4 ;resonance
imode = p5 ;mode
asig vco 0.2, 220, 1
afilt bqrez asig, kfco, kres, imode
asig balance afilt, asig
outs asig, asig

endin

</CsInstruments>
<CsScore>
;sine wave
f 1 0 16384 10 1

i 1 0 3 1 0 ; low pass
i 1 + 3 30 0 ; low pass
i 1 + 3 1 1 ; high pass
i 1 + 3 30 1 ; high pass

e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*biquad, moogvcf, rezzy*

## Credits

Author: Matt Gerassimoff  
New in version 4.32  
Written in November 2002.

## butbp

butbp — Same as the `butterbp` opcode.

### Description

Same as the *butterbp* opcode.

### Syntax

```
ares butbp asig, kfreq, kband [, iskip]
```

## butbr

butbr — Same as the *butterbr* opcode.

## Description

Same as the *butterbr* opcode.

## Syntax

```
ares butbr asig, kfreq, kband [, iskip]
```

## buthp

buthp — Same as the *butterhp* opcode.

### Description

Same as the *butterhp* opcode.

### Syntax

```
ares buthp asig, kfreq [, iskip]
```

```
ares buthp asig, afreq [, iskip]
```

# butlp

butlp — Same as the *butterlp* opcode.

## Description

Same as the *butterlp* opcode.

## Syntax

```
ares butlp asig, kfreq [, iskip]
```

```
ares butlp asig, afreq [, iskip]
```

# butterbp

butterbp — A band-pass Butterworth filter.

## Description

Implementation of a second-order band-pass Butterworth filter. This opcode can also be written as *butbp*.

## Syntax

```
ares butterbp asig, xfreq, xband [, iskip]
```

## Initialization

*iskip* (optional, default=0) -- Skip initialization if present and non-zero.

## Performance

These filters are Butterworth second-order IIR filters. They are slightly slower than the original filters in Csound, but they offer an almost flat passband and very good precision and stopband attenuation.

*asig* -- Input signal to be filtered.

*xfreq* -- Cutoff or center frequency for each of the filters.

*xband* -- Bandwidth of the bandpass and bandreject filters.

## Examples

Here is an example of the butterbp opcode. It uses the file *butterbp.csd* [examples/butterbp.csd].

### Example 97. Example of the butterbp opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o butterbp.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; White noise signal
asig rand 0.6
```

```
    outs asig, asig

endin

instr 2 ;filtered noise

asig rand 1
abp butterbp asig, 2000, 100 ;passing only 1950 to 2050 Hz
    outs abp, abp

endin

</CsInstruments>
<CsScore>

i 1 0 2
i 2 2.5 2
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*butterbr, butterhp, butterlp*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

Existed in 3.30

Audio rate parameters introduced in version 6.02

October 2013.

# butterbr

butterbr — A band-reject Butterworth filter.

## Description

Implementation of a second-order band-reject Butterworth filter. This opcode can also be written as *butbr*.

## Syntax

```
ares butterbr asig, xfreq, xband [, iskip]
```

## Initialization

*iskip* (optional, default=0) -- Skip initialization if present and non-zero.

## Performance

These filters are Butterworth second-order IIR filters. They are slightly slower than the original filters in Csound, but they offer an almost flat passband and very good precision and stopband attenuation.

*asig* -- Input signal to be filtered.

*xfreq* -- Cutoff or center frequency for each of the filters.

*xband* -- Bandwidth of the bandpass and bandreject filters.

## Examples

Here is an example of the butterbr opcode. It uses the file *butterbr.csd* [examples/butterbr.csd].

### Example 98. Example of the butterbr opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o butterbr.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; White noise

asig rand 0.5
```

```
    outs asig, asig

endin

instr 2 ; filtered noise

asig rand 0.7
abr butterbr asig, 3000, 2000 ;center frequency = 3000, bandwidth = +/- (2000)/2, so 2000-4000
    outs abr, abr

endin

</CsInstruments>
<CsScore>

i 1 0 2
i 2 2.5 2

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*butterbp, butterhp, butterlp*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

Existed in 3.30

Audio rate parameters introduced in version 6.02

October 2013.

# butterhp

butterhp — A high-pass Butterworth filter.

## Description

Implementation of second-order high-pass Butterworth filter. This opcode can also be written as *buthp*.

## Syntax

```
ares butterhp asig, kfreq [, iskip]
ares butterhp asig, afreq [, iskip]
```

## Initialization

*iskip* (optional, default=0) -- Skip initialization if present and non-zero.

## Performance

These filters are Butterworth second-order IIR filters. They are slightly slower than the original filters in Csound, but they offer an almost flat passband and very good precision and stopband attenuation.

*asig* -- Input signal to be filtered.

*kfreq/afreq* -- Cutoff or center frequency for each of the filters.

## Examples

Here is an example of the butterhp opcode. It uses the file *butterhp.csd* [examples/butterhp.csd].

### Example 99. Example of the butterhp opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o butterhp.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; White noise
asig rand 0.5
```

```
outs asig, asig

endin

instr 2 ; filtered noise

asig rand 0.6
ahp butterhp asig, 500 ;pass frequencies above 500 Hz
outs ahp, ahp

endin

</CsInstruments>
<CsScore>

i 1 0 2
i 2 2.5 2
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*butterbp, butterbr, butterlp*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

Existed in 3.30

Audio rate parameters introduced in version 6.02

October 2013.

# butterlp

butterlp — A low-pass Butterworth filter.

## Description

Implementation of a second-order low-pass Butterworth filter. This opcode can also be written as *butlp*.

## Syntax

```
ares butterlp asig, kfreq [, iskip]
ares butterlp asig, afreq [, iskip]
```

## Initialization

*iskip* (optional, default=0) -- Skip initialization if present and non-zero.

## Performance

These filters are Butterworth second-order IIR filters. They are slightly slower than the original filters in Csound, but they offer an almost flat passband and very good precision and stopband attenuation.

*asig* -- Input signal to be filtered.

*kfreq/afreq* -- Cutoff or center frequency for each of the filters.

## Examples

Here is an example of the butterlp opcode. It uses the file *butterlp.csd* [examples/butterlp.csd].

### Example 100. Example of the butterlp opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o butterlp.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; White noise signal
asig rand 0.5
```

```
    outs asig, asig

endin

instr 2 ; filtered noise

asig rand 0.7
alp butterlp asig, 1000 ;cutting frequencies above 1 KHz
    outs alp, alp

endin

</CsInstruments>
<CsScore>

i 1 0 2
i 2 2.5 2
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*butterbp, butterbr, butterhp*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

Existed in 3.30

Audio rate parameters introduced in version 6.02

October 2013.

# button

button — Sense on-screen controls.

## Description

Sense on-screen controls. Requires Winsound or TCL/TK.

## Syntax

```
kres button knum
```

## Performance

Note that this opcode is not available on Windows due to the implimentation of pipes on that system

*kres* -- value of the button control. If the button has been pushed since the last k-period, then return 1, otherwise return 0.

*knun* -- the number of the button. If it does not exist, it is made on-screen at initialization.

## Examples

Here is a simple example of the checkbox opcode. It uses the file *checkbox.csd* [examples/checkbox.csd].

### Example 101. Simple example of the checkbox opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o checkbox.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 10
nchnls = 2

instr 1
; Get the value from the checkbox.
k1 checkbox 1

; If the checkbox is selected then k2=440, otherwise k2=880.
k2 = (k1 == 0 ? 440 : 880)

a1 oscil 10000, k2, 1
outs a1, a1
kq button 1
if kq = 1 then
```

```
        exitnow
    endif
endin

</CsInstruments>
<CsScore>

; sine wave.
f 1 0 32768 10 1

i 1 0 1000
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*checkbox*

## Credits

Author: John fitch  
University of Bath, Codemist. Ltd.  
Bath, UK  
September 2000

New in Csound version 4.08

# buzz

`buzz` — Output is a set of harmonically related sine partials.

## Description

Output is a set of harmonically related sine partials.

## Syntax

```
ares buzz xamp, xcps, knh, ifn [, iphs]
```

## Initialization

*ifn* -- table number of a stored function containing a sine wave. A large table of at least 8192 points is recommended.

*iphs* (optional, default=0) -- initial phase of the fundamental frequency, expressed as a fraction of a cycle (0 to 1). A negative value will cause phase initialization to be skipped. The default value is zero

## Performance

*xamp* -- amplitude

*xcps* -- frequency in cycles per second

The *buzz* units generate an additive set of harmonically related cosine partials of fundamental frequency *xcps*, and whose amplitudes are scaled so their summation peak equals *xamp*. The selection and strength of partials is determined by the following control parameters:

*knh* -- total number of harmonics requested. New in Csound version 3.57, *knh* defaults to one. If *knh* is negative, the absolute value is used.

*buzz* and *gbuzz* are useful as complex sound sources in subtractive synthesis. *buzz* is a special case of the more general *gbuzz* in which  $klh = kmul = 1$ ; it thus produces a set of *knh* equal-strength harmonic partials, beginning with the fundamental. (This is a band-limited pulse train; if the partials extend to the Nyquist, i.e.  $knh = \text{int}(sr / 2 / \text{fundamental freq.})$ , the result is a real pulse train of amplitude *xamp*.)

Although *knh* may be varied during performance, its internal value is necessarily integer and may cause “pops” due to discontinuities in the output. *buzz* can be amplitude- and/or frequency-modulated by either control or audio signals.

N.B. This unit has its analog in *GENII*, in which the same set of cosines can be stored in a function table for sampling by an oscillator. Although computationally more efficient, the stored pulse train has a fixed spectral content, not a time-varying one as above.

## Examples

Here is an example of the *buzz* opcode. It uses the file *buzz.csd* [examples/buzz.csd].

## Example 102. Example of the buzz opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o buzz.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kcps = 110
ifn = 1

knh line p4, p3, p5
asig buzz 1, kcps, knh, ifn
outs asig, asig
endin

</CsInstruments>
<CsScore>

;sine wave.
f 1 0 16384 10 1

i 1 0 3 20 20
i 1 + 3 3 3
i 1 + 3 10 1
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*gbuzz*

## Credits

September 2003. Thanks to Kanata Motohashi for correcting the mentions of the *kmul* parameter.

# c2r

c2r — Real to complex format conversion.

## Description

Converts a complex-valued input array into a real-valued array, by removing its imaginary part. The output array will be half the size of the input. This is a utility operation to facilitate complex-valued operations where the result is purely real.

## Syntax

```
kout[] c2r kin[]
```

## Performance

*kout[]* -- output array containing the real-valued output. It will be created if it does not exist.

*kin[]* -- input array containing the complex-valued real-imaginary input.

## Examples

Here is an example of the c2r opcode. It uses the file *c2r.csd* [examples/c2r.csd].

### Example 103. Example of the c2r opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-d -o dac
</CsOptions>
<CsInstruments>
ksmps = 64

instr 1
  ifftsize = 1024
  kcnt init 0
  kIn[] init ifftsize
  kOut[] init ifftsize

  a1 oscili 0dbfs/2, 440

  if kcnt >= ifftsize then
    kCmplx[] r2c kIn
    kSpec[] fft kCmplx
    kCmplx fftinv kSpec
    kOut c2r kCmplx
    kcnt = 0
  endif

  kIn[] shiftin a1
  a2 shiftout kOut
  kcnt += ksmps
```

```
    out a2  
endin  
</CsInstruments>  
<CsScore>  
i1 0 10  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

Vectorial opcodes, array opcodes

## Credits

Author: Victor Lazzarini  
NUI Maynooth  
2014

New in version 6.04

# cabasa

cabasa — Semi-physical model of a cabasa sound.

## Description

*cabasa* is a semi-physical model of a cabasa sound. It is one of the PhISEM percussion opcodes. PhISEM (Physically Informed Stochastic Event Modeling) is an algorithmic approach for simulating collisions of multiple independent sound producing objects.

## Syntax

```
ares cabasa iamp, idettack [, inum] [, idamp] [, imaxshake]
```

## Initialization

*iamp* -- Amplitude of output. Note: As these instruments are stochastic, this is only a approximation.

*idettack* -- period of time over which all sound is stopped

*inum* (optional) -- The number of beads, teeth, bells, timbrels, etc. If zero, the default value is 512.

*idamp* (optional) -- the damping factor, as part of this equation:

$$\text{damping\_amount} = 0.998 + (\text{idamp} * 0.002)$$

The default *damping\_amount* is 0.997 which means that the default value of *idamp* is -0.5. The maximum *damping\_amount* is 1.0 (no damping). This means the maximum value for *idamp* is 1.0.

The recommended range for *idamp* is usually below 75% of the maximum value.

*imaxshake* (optional) -- amount of energy to add back into the system. The value should be in range 0 to 1.

## Examples

Here is an example of the cabasa opcode. It uses the file *cabasa.csd* [examples/cabasa.csd].

### Example 104. Example of the cabasa opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o cabasa.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
```

```
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

inum = p4
idamp = p5
asig cabasa 0.9, 0.01, inum, idamp
outs asig, asig

endin

</CsInstruments>
<CsScore>

i1 1 1 48 .95
i1 + 1 1000 .5

e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*crunch, sandpaper, sekere, stix*

## Credits

Author: Perry Cook, part of the PhISEM (Physically Informed Stochastic Event Modeling)

Adapted by John ffitch

University of Bath, Codemist Ltd.

Bath, UK

New in Csound version 4.07

Added notes by Rasmus Ekman on May 2002.

# cauchy

cauchy — Cauchy distribution random number generator.

## Description

Cauchy distribution random number generator. This is an x-class noise generator.

## Syntax

```
ares cauchy kalpha
ires cauchy kalpha
kres cauchy kalpha
```

## Performance

*kalpha* -- controls the spread from zero (big *kalpha* = big spread). Outputs both positive and negative numbers.

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the cauchy opcode. It uses the file *cauchy.csd* [examples/cauchy.csd].

### Example 105. Example of the cauchy opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o cauchy.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; every run time same values

kalpha cauchy 100
```

```

    printk .2, kalpha ; look
    aout oscili 0.8, 440+kalpha, 1 ; & listen
    outs aout, aout
    endin

    instr 2 ; every run time different values

    seed 0
    kalpha cauchy 100
    printk .2, kalpha ; look
    aout oscili 0.8, 440+kalpha, 1 ; & listen
    outs aout, aout
    endin
</CsInstruments>
<CsScore>
; sine wave
f 1 0 16384 10 1

i 1 0 2
i 2 3 2
e

</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like:

```

i 1 time 0.00267: -0.20676
i 1 time 0.20267: -0.28814
i 1 time 0.40267: 0.61651
i 1 time 0.60267: -18.18552
i 1 time 0.80267: 1.18140
i 1 time 1.00000: -0.75432
i 1 time 1.20267: -0.02002
i 1 time 1.40267: 0.01785
i 1 time 1.60267: -0.48834
i 1 time 1.80000: 9.69401
i 1 time 2.00000: -0.41257
Seeding from current time 3112109827
i 2 time 3.00267: -0.46887
i 2 time 3.20267: 0.06189
i 2 time 3.40267: -0.40303
i 2 time 3.60000: 0.89312
i 2 time 3.80267: -0.40374
i 2 time 4.00000: 0.86557
i 2 time 4.20000: 0.09192
i 2 time 4.40267: -0.16748
i 2 time 4.60267: 0.30133
i 2 time 4.80267: 0.31657
i 2 time 5.00000: 0.44681

```

## See Also

*seed, betarand, bexprnd, exprand, gauss, linrand, pcauchy, poisson, trirand, unirand, weibull*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

Existed in 3.30

# cauchy

cauchy — Cauchy distribution random number generator with interpolation.

## Description

Cauchy distribution random number generator with controlled interpolation between values. This is an x-class noise generator.

## Syntax

```
ares cauchy klambda, xamp, xcps
```

```
ires cauchy klambda, xamp, xcps
```

```
kres cauchy klambda, xamp, xcps
```

## Performance

*klambda* -- controls the spread from zero (big *klambda* = big spread). Outputs both positive and negative numbers.

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

*xamp* -- range over which random numbers are distributed.

*xcps* -- the frequency which new random numbers are generated.

## Examples

Here is an example of the cauchy opcode. It uses the file *cauchy.csd* [examples/cauchy.csd].

### Example 106. Example of the cauchy opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o exprand.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
```

```
nchnls = 2
odbfs = 1

instr 1
klambda cauchy 100, 1, 3
  printk2 klambda ; look
aout oscili 0.8, 440+klambda, 1 ; & listen
  outs aout, aout
endin

</CsInstruments>
<CsScore>
; sine wave
f 1 0 16384 10 1

i 1 0 4
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*cauchy*

## Credits

Author: John ffitch  
Bath  
May 2011  
New in version 5.14

# ceil

ceil — Returns the smallest integer not less than  $x$

## Description

Returns the smallest integer not less than  $x$

## Syntax

`ceil(x)` (init-, control-, or audio-rate arg allowed)

`ceil(k/i[])` (k- or i-arrays )

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

## Examples

Here is an example of the ceil opcode. It uses the file `ceil.csd` [examples/ceil.csd].

### Example 107. Example of the ceil opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too

</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

instr 1

inum = p4
iceil = ceil(inum)
print iceil

endin

</CsInstruments>
<CsScore>

i 1 0 0 1
i . . . 0.999999
i . . . 0.000001
i . . . 0
i . . . -0.0000001
i . . . -0.9999999

```

```
i . . . -1  
e  
</CsScore>  
</CsoundSynthesizer>
```

Its output should include lines like:

```
instr 1:  ceil = -1.000  
instr 1:  ceil = 1.000  
instr 1:  ceil = 1.000  
instr 1:  ceil = 1.000  
instr 1:  ceil = 0.000  
instr 1:  ceil = 0.000  
instr 1:  ceil = 0.000
```

## See Also

*abs, exp, int, log, log10, i, sqrt*

## Credits

Author: Istvan Varga  
New in Csound 5  
2005

# cell

cell — Cellular Automaton

## Description

One-Dimensional Cellular Automaton. This opcode is the modified version of *vcella* by Gabriel Maldonado.

## Syntax

```
cell ktrig, kreinit, ioutFunc, initStateFunc, iRuleFunc, ielements
```

## Initialization

*ioutFunc* -- number of the table where the state of each cell is stored.

*initStateFunc* -- number of the table containing the initial states of cells.

*iRuleFunc* -- number of a lookup table containing the 8-bit rule.

*ielements* -- total number of cells in a row.

## Performance

*ktri* -- trigger signal. Each time it is non-zero, a new generation of cells is evaluated.

*kreinit* -- reset signal. Each time it is non-zero, state of all cells is forced to be that of *initStateFunc*.

*cell* models a classical 1D cellular automaton and stores the state of each cell in the table identified by *ioutFunc*.

*initStateFunc* is an input vector containing the initial value of the row of cells, while *iRuleFunc* is an input vector containing the chosen rule in the binary form (least significant bit first).

A new generation of cells is evaluated each time *ktrig* contains a non-zero value. Also the status of all cells can be forced to assume the status corresponding to the contents of *initStateFunc* each time *kreinit* contains a non-zero value.

Note that each cell is supposed to be in one of two possible states (1 = "alive", 0 = "dead"), although fractional values should work too, because truncation is used.

## Examples

Here is a simple example of the cell opcode. It uses the file *cell.csd* [examples/cell.csd].

### Example 108. A simple example of the cell opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
;-odac      -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o cell.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
; cell.csd by Gleb Rogozinsky

sr      = 44100
kr      = 4410
ksmps   = 10
nchnls  = 1
0dbfs   = 1

; Cellular automaton-driven synthesis in spectral domain
instr 1

  iatt    = p4                ; envelope attack time
  isus    = p5                ; envelope sustain time
  irel    = p6                ; envelope release time
  ivol    = p7                ; overall volume

; create some white noise
  asig    rand      0.8

; spectral analysis of asig
  fsig    pvsanal   asig, 2048, 1024, 2048, 0 ; get a vector of magnitudes

; calculate cellular automaton state
  kfreq   line      50, 5, 1                ; variable CA triggering frequency
  ktrig   metro     kfreq                    ; trigger the CA to update cells
          cell      ktrig, 0, 3, 1, 2, 2048 ; cells are written into ftable 3

; use current row of cells in spectral domain
  fmas    pvstencil  fsig, 0, 1, 3          ; apply spectral mask
  aout    pvsynth   fmas                    ; reconstruct time signal

; apply envelope and out signal
  kenv    expseg    .001, iatt, 1, isus, 1, irel, .001
          out      aout*kenv*ivol
endin

</CsInstruments>
<CsScore>

; This example uses one-dimensional cellular automaton
; to produce structures in spectral domain

; We have to prepare initial row of cells.
; One alive cell is enough to produce a simple fractal,
; so two alive cells will make structure more sophisticated
f1 0 2048 7 0 150 0 0 1 1 1 0 0 45 0 0 1 1 1 0 0

; The CA rule is used as follows:
; the states (values) of each cell are summed with their neighboring cells.
; Each sum is used as an index to read a next state of cell
; from the rule table.
; Let us try rule # 129 (LSB binary 1 0 0 0 0 0 0 1).
; This rule will produce a fractal structure for single active cell
; For more rules see http://mathworld.wolfram.com/ElementaryCellularAutomaton.html
f2 0 8 -2 1 0 0 0 0 0 0 1
; Try some different rules i.E. 254 (0 1 1 1 1 1 1 1) or 169 (1 0 0 1 0 1 0 1)

```

```
; Prepare the output table of ielements size  
f3 0 2048 10 0  
  
; Time to make it sound!  
i1 0 13 0.3 7 3 1  
e  
  
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Author: Gleb Rogozinsky  
October 2011

New in Csound version 5.16.6

# cent

cent — Calculates a factor to raise/lower a frequency by a given amount of cents.

## Description

Calculates a factor to raise/lower a frequency by a given amount of cents.

## Syntax

```
cent(x)
```

This function works at a-rate, i-rate, and k-rate.

## Initialization

*x* -- a value expressed in cents.

## Performance

The value returned by the *cent* function is a factor. You can multiply a frequency by this factor to raise/lower it by the given amount of cents.

## Examples

Here is an example of the cent opcode. It uses the file *cent.csd* [examples/cent.csd].

### Example 109. Example of the cent opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o cent.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; original pitch

iroot = 440 ; root note = A (440 Hz)

print iroot ;print out

asig oscili 0.6, iroot, 1
```

```
    outs asig, asig

endin

instr 2

iroot  = 440 ; root note = A (440 Hz)
icents = p4 ; change root note by 300 and 1200 cents

ifactor = cent(icents) ; calculate new note
inew    = iroot * ifactor

print iroot ; Print all
print ifactor
print inew

asig oscili 0.6, inew, 1
    outs asig, asig

endin

</CsInstruments>
<CsScore>
; sine wave
f1 0 32768 10 1

i 1 0 2 ;no change
i 2 2.5 2 300 ;note = C above A
i 2 5 2 1200 ;1 octave higher

e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like:

```
instr 1: iroot = 440.000

instr 2: iroot = 440.000
instr 2: ifactor = 1.189
instr 2: inew = 523.251

instr 2: iroot = 440.000
instr 2: ifactor = 2.000
instr 2: inew = 880.000
```

## See Also

*db, octave, semitone*

New in version 4.16

# centroid

centroid — Calculate the spectral centroid of a signal.

## Description

Calculate the spectral centroid of an audio signal on a given trigger.

## Syntax

```
kcent centroid asig, ktrig, ifftsize
```

## Initialization

*ifftsize* -- fftsize. Non pow-of-two values are converted to the next pow-of-two not smaller than ifftsize.

## Performance

*kcent* -- the spectral centroid in Hz

*asig* -- an input audio signal

*ktrig* -- 1 to calculate a new centroid, 0 to skip the process (and output previous value).

## Examples

Here is an example of the centroid opcode. It uses the file *centroid.csd* [examples/centroid.csd].

### Example 110. Example of the centroid opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o centroid.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine ftgen 0, 0, 16384, 10, 1

instr 1

ktrig init 1 ;calculate centroid
a1 oscil3 0.5, p4, giSine ;of the sine wave
k1 centroid a1, ktrig, 16384
```

```
asig oscil3 0.5, k1, giSine
printk2 k1 ;print & compare:
outs a1, asig ;left = original, right = centroid signal

endin
</CsInstruments>
<CsScore>

i1 0 2 20
i1 + 2 200
i1 + 2 2000
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvscent*

## Credits

Author: V Lazzarini;  
August 2012

# ceps

ceps — Calculate the cepstrum of an array input, optionally liftering coefficients.

## Description

## Syntax

```
keps[] ceps kmags[][], icoefs]
```

## Initialization

*icoefs* -- the number of retained coefficients in the cepstrum output. By default, no coefficients are liftered.

## Performance

*keps[]* -- the cepstrum output, an array of size N+1, where N is a power of two.

*kmags[]* -- an input array containing N+1 magnitudes.

## Examples

Here is an example of the use of the *ceps* opcode. It uses the file *ceps.csd* [examples/ceps.csd].

### Example 111. Example of the *ceps* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>

instr 1

a1 diskin "fox.wav",1,0,1
k1 randh 80, 2.5
a2 vco2 1, 220+k1
kfsig[] init 1026
fsig pvsanal a1,1024,256,1024,1
fsig2 pvsanal a2,1024,256,1024,1
kf pvs2tab kfsig,fsig
keps[] ceps c2r(kfsig),30
kenv[] cepsinv keps
fenv tab2pvs r2c(kenv)
fvoc pvsfilter fsig2, fenv, 1
asig pvsynth fvoc

    out asig
endin

</CsInstruments>
```

```
<CsScore>  
i1 0 60  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*pvsanal, pvsynth, pvsadsyn, pvspitch*

## Credits

Author: Victor Lazzarini  
2014

# cepsinv

`cepsinv` — Calculate the inverse cepstrum of an array.

## Description

## Syntax

```
kenv cepsinv keps[]
```

## Performance

*keps[]* -- the cepstrum input, an array of size N+1, where N is power-of-two, containing the cepstral coefficients.

*kenv* -- the inverse cepstrum (spectral envelope), an array of N+1 magnitudes.

## Examples

Here is an example of the use of the *cepsinv* opcode. It uses the file *cepsinv.csd* [examples/cepsinv.csd].

### Example 112. Example of the *cepsinv* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>

instr 1

a1 diskin "fox.wav",1,0,1
k1 randh 80, 2.5
a2 vco2 8, 220+k1
fsig pvsanal a1,1024,256,1024,1
fsig2 pvsanal a2,1024,256,1024,1
keps[] pvsceps fsig,30
kenv[] cepsinv keps
fenv tab2pvs r2c(kenv)
fvoc pvsfilter fsig2, fenv, 1
asig pvsynth fvoc

    out asig
endin

</CsInstruments>
<CsScore>
i1 0 60
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
2014

# cggoto

cggoto — Conditionally transfer control on every pass.

## Description

Transfer control to *label* on every pass. (Combination of *cigoto* and *ckgoto*)

## Syntax

```
cggoto condition, label
```

where *label* is in the same instrument block and is not an expression, and where *condition* uses one of the Relational operators (<, =, <=, ==, !=) (and = for convenience, see also under *Conditional Values*).

## Examples

Here is an example of the cggoto opcode. It uses the file *cggoto.csd* [examples/cggoto.csd].

### Example 113. Example of the cggoto opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Audio out   Audio in   No messages
-odac         -iadc      -d      ;;RT audio I/O
; -o cggoto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  i1 = p4

  ; If i1 is equal to one, play a high note.
  ; Otherwise play a low note.
  cggoto (i1 == 1), highnote

lownote:
  a1 oscil 10000, 220, 1
  goto playit

highnote:
  a1 oscil 10000, 440, 1
  goto playit

playit:
  out a1
endin
```

```
</CsInstruments>
<CsScore>

; Table #1: a simple sine wave.
f 1 0 32768 10 1

; Play lownote for one second.
i 1 0 1 1
; Play highnote for one second.
i 1 1 1 2
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*cigoto, ckgoto, cngoto, if, igoto, kgoto, tigo, timeout*

## Credits

Example written by Kevin Conder.

# chanctrl

chanctrl — Get the current value of a MIDI channel controller.

## Description

Get the current value of a controller and optionally map it onto specified range.

## Syntax

```
ival chanctrl ichnl, ictlno [, ilow] [, ihigh]
kval chanctrl ichnl, ictlno [, ilow] [, ihigh]
```

## Initialization

*ichnl* -- the MIDI channel (1-16).

*ictlno* -- the MIDI controller number (0-127).

*ilow*, *ihigh* -- low and high ranges for mapping

## Examples

Here is an example of the chanctrl opcode. It uses the file *chanctrl.csd* [examples/chanctrl.csd].

### Example 114. Example of the chanctrl opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  No messages MIDI in
-odac          -d          -M0   ;;RT audio I/O with MIDI in
;-iadc         ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o chanctrl.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; press your midi keyboard and move your midi controller to see result

ichnl = 1 ;MIDI note inputs on channel 1
ictlno = 7 ;use midi volume controller
kch chanctrl ichnl, 7 ;to control amplitude of oscil
printk2 kch

asig oscil kch*(1/127), 220, 1
outs asig, asig
```

```
endin

</CsInstruments>
<CsScore>
;Dummy f-table to give time for real-time MIDI events
f 0 30
;sine wave.
f 1 0 16384 10 1
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Mike Berry  
Mills College  
May, 1997

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# changed

changed — k-rate signal change detector.

## Description

This opcode outputs a trigger signal that informs when any one of its k-rate arguments has changed. Useful with valuator widgets or MIDI controllers.

## Syntax

```
ktrig changed kvar1 [, kvar2,..., kvarN]
```

## Performance

*ktrig* - Outputs a value of 1 when any of the k-rate signals has changed, otherwise outputs 0.

*kvar1* [, *kvar2*,..., *kvarN*] - k-rate variables to watch for changes.



### Note

If any of the kvars is non-zero at the first k-cycle changed is called it will report a change. For more consistent behaviour use *changed2*.

## Examples

Here is an example of the changed opcode. It uses the file *changed.csd* [examples/changed.csd].

### Example 115. Example of the changed opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o changed.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

instr 1

ksig oscil 2, 0.5, 1
kint = int(ksig)
ktrig changed kint
      printk 0.2, kint
      printk2 ktrig

endin

</CsInstruments>
```

```
<CsScore>
f 1 0 1024 10 1
i 1 0 20

e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like:

```
i 1 time 0.00267: 0.00000
il 0.00000
il 1.00000
il 0.00000
i 1 time 0.20267: 1.00000
i 1 time 0.40267: 1.00000
il 1.00000
il 0.00000
i 1 time 0.60267: 1.00000
i 1 time 0.80267: 1.00000
il 1.00000
il 0.00000
i 1 time 1.00000: 0.00000
il 1.00000
il 0.00000
i 1 time 1.20267: -1.00000
i 1 time 1.40267: -1.00000
il 1.00000
il 0.00000
i 1 time 1.60267: -1.00000
i 1 time 1.80000: -1.00000
il 1.00000
il 0.00000
i 1 time 2.00000: -0.00000
il 1.00000
il 0.00000
.....
```

## Credits

Written by Gabriel Maldonado.

Example written by Andrés Cabrera.

New in Csound 7.

# changed2

changed2 — k-rate signal change detector.

## Description

This opcode outputs a trigger signal that informs when any one of its k-rate arguments has changed, or a value in an array. Useful with valuator widgets or MIDI controllers.

## Syntax

```
ktrig changed2 kvar1 [, kvar2,..., kvarN]
ktrig changed2 karr[]
ktrig changed2 aarr[]
```

## Performance

*ktrig* - Outputs a value of 1 when any of the k-rate signals has changed, or any value in the array, otherwise outputs 0.

*kvar1* [, *kvar2*,..., *kvarN*] - k-rate variables to watch for changes.

*xarray* - any array type.

Unlike the opcode *changed* this opcode will ever report the first cycle as a change.

## Examples

Here are two examples of the *changed2* opcode. They use the files *changed2.csd* [examples/changed2.csd] and *changed2a.csd* [examples/changed2a.csd].

### Example 116. Example of the *changed2* opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o changed.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

instr 1

ksig oscil 2, 0.5, 1
kint = int(ksig)
ktrig changed2 kint
printk 0.2, kint
```

```

    printk2 ktrig

endin

</CsInstruments>
<CsScore>
f 1 0 1024 10 1
i 1 0 20

e
</CsScore>
</CsSoundSynthesizer>

```

Its output should include lines like:

```

i 1 time 0.00267: 0.00000
il 0.00000
il 1.00000
il 0.00000
i 1 time 0.20267: 1.00000
i 1 time 0.40267: 1.00000
il 1.00000
il 0.00000
i 1 time 0.60267: 1.00000
i 1 time 0.80267: 1.00000
il 1.00000
il 0.00000
i 1 time 1.00000: 0.00000
il 1.00000
il 0.00000
i 1 time 1.20267: -1.00000
i 1 time 1.40267: -1.00000
il 1.00000
il 0.00000
i 1 time 1.60267: -1.00000
i 1 time 1.80000: -1.00000
il 1.00000
il 0.00000
i 1 time 2.00000: -0.00000
il 1.00000
il 0.00000
.....

```

### Example 117. Example of the changed2 opcode.

```

<CsSoundSynthesizer>

<CsOptions>
-n
</CsOptions>
<CsInstruments>
gkArray[][] init 2,3
gkArray array 1,2,3,7,6,5

instr 1
k1 changed2 gkArray
if k1==1 then
    printks "An element in the array changed", 0
endif
endin

instr 2; change value of channel 'step1'
gkArray[1][0] = 3

```

```
    endin  
  
    </CsInstruments>  
    <CsScore>  
    i1 0 100  
    i2 4 .1  
    </CsScore>  
  
    </CsoundSynthesizer>
```

## Credits

Written by John ffitth.

Example written by Andrés Cabrera.

New in Csound 6.7

# chani

chani — Reads data from the software bus

## Description

Reads data from a channel of the inward software bus.

## Syntax

```
kval chani kchan
```

```
aval chani kchan
```

## Performance

*kchan* -- a positive integer that indicates which channel of the software bus to read

Note that the inward and outward software busses are independent, and are not mixer busses. Also the k-rate and a-rate busses are independent. The last value remains until a new value is written. There is no imposed limit to the number of busses but they use memory so small numbers are to be preferred.

## Example

The example shows the software bus being used as an asynchronous control signal to select a filter cutoff. It assumes that an external program that has access to the API is feeding the values

```
sr = 44100
kr = 100
ksmps = 1

instr 1
  kc  chani      1
  a1  oscil      p4, p5, 100
  a2  lowpass2   a1, kc, 200
  out          a2
endin
```

## Credits

Author: John fitch  
2005

New in Csound 5.00

# chano

chano — Send data to the outwards software bus

## Description

Send data to a channel of the outward software bus.

## Syntax

```
chano kval, kchan
```

```
chano aval, kchan
```

## Performance

*xval* --- value to transmit

*kchan* -- a positive integer that indicates which channel of the software bus to write

Note that the inward and outward software busses are independent, and are not mixer busses. Also the k-rate and a-rate busses are independent. The last value remains until a new value is written. There is no imposed limit to the number of busses but they use memory so small numbers are to be preferred.

## Example

The example shows the software bus being used as an output audio channel. It assumes that an external program that has access to the API is receiving the values.

```
sr = 44100
kr = 100
ksmps = 1

instr 1
  a1 oscil      p4, p5, 100
     chano     1, a1
endin
```

## Credits

Author: John ffitch  
2005

New in Csound 5.00

# cbrt

cbrt — Cubic root function.

## Description

Returns the value of the cubic root of an argument.

## Syntax

```
iRes[] cbrt iArg
```

```
kRes[] cbrt kArg
```

## Initialization

*iArg[]* -- the argument.

## Performance

*kArg[]* -- the argument.

## Examples

Here is an example of the `cbrt` opcode. It uses the file `cbrt.csd` [examples/cbrt.csd].

### Example 118. Example of the `cbrt` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>

instr 1

iArg[] fillarray 1,2,3
iRes[] cbrt iArg
ik init 0

while ik < lenarray(iRes) do
  print iRes[ik]
  ik += 1
od

endin

</CsInstruments>
<CsScore>
i1 0 0
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
2017

# chebyshevpoly

chebyshevpoly — Efficiently evaluates the sum of Chebyshev polynomials of arbitrary order.

## Description

The *chebyshevpoly* opcode calculates the value of a polynomial expression with a single a-rate input variable that is made up of a linear combination of the first N Chebyshev polynomials of the first kind. Each Chebyshev polynomial,  $T_n(x)$ , is weighted by a k-rate coefficient,  $kn$ , so that the opcode is calculating a sum of any number of terms in the form  $kn*T_n(x)$ . Thus, the *chebyshevpoly* opcode allows for the waveshaping of an audio signal with a *dynamic* transfer function that gives precise control over the harmonic content of the output.

## Syntax

```
aout chebyshevpoly ain, k0 [, k1 [, k2 [...]]]
```

## Performance

*ain* -- the input signal used as the independent variable of the Chebyshev polynomials ("x").

*aout* -- the output signal ("y").

*k0, k1, k2, ...* -- k-rate multipliers for each Chebyshev polynomial.

This opcode is very useful for dynamic waveshaping of an audio signal. Traditional waveshaping techniques utilize a lookup table for the transfer function -- usually a sum of Chebyshev polynomials. When a sine wave at full-scale amplitude is used as an index to read the table, the precise harmonic spectrum as defined by the weights of the Chebyshev polynomials is produced. A dynamic spectrum is achieved by varying the amplitude of the input sine wave, but this produces a non-linear change in the spectrum.

By directly calculating the Chebyshev polynomials, the *chebyshevpoly* opcode allows more control over the spectrum and the number of harmonic partials added to the input can be varied with time. The value of each  $kn$  coefficient directly controls the amplitude of the  $n$ th harmonic partial if the input *ain* is a sine wave with amplitude = 1.0. This makes *chebyshevpoly* an efficient additive synthesis engine for N partials that requires only one oscillator instead of N oscillators. The amplitude or waveform of the input signal can also be changed for different waveshaping effects.

If we consider the input parameter *ain* to be "x" and the output *aout* to be "y", then the *chebyshevpoly* opcode calculates the following equation:

$$y = k0*T0(x) + k1*T1(x) + k2*T2(x) + k3*T3(x) + \dots$$

where the  $T_n(x)$  are defined by the recurrence relation

$$\begin{aligned} T0(x) &= 1, \\ T1(x) &= x, \\ Tn(x) &= 2x*T[n-1](x) - T[n-2](x) \end{aligned}$$

More information about Chebyshev polynomials can be found on Wikipedia at [http://en.wikipedia.org/wiki/Chebyshev\\_polynomial](http://en.wikipedia.org/wiki/Chebyshev_polynomial) [http://en.wikipedia.org/wiki/Chebyshev\_polynomial]

## See Also

*polynomial*, *mac maca*

## Examples

Here is an example of the `chebyshevpoly` opcode. It uses the file `chebyshevpoly.csd` [examples/chebyshevpoly.csd].

### Example 119. Example of the `chebyshevpoly` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o chebyshevpoly.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

; time-varying mixture of first six harmonics
instr 1
; According to the GEN13 manual entry,
; the pattern + - - + + - - for the signs of
; the chebyshev coefficients has nice properties.

; these six lines control the relative powers of the harmonics
k1      line      1.0, p3, 0.0
k2      line      -0.5, p3, 0.0
k3      line      -0.333, p3, -1.0
k4      line      0.0, p3, 0.5
k5      line      0.0, p3, 0.7
k6      line      0.0, p3, -1.0

; play the sine wave at a frequency of 256 Hz with amplitude = 1.0
ax      oscili    1, 256, 1

; waveshape it
ay      chebyshevpoly ax, 0, k1, k2, k3, k4, k5, k6

; avoid clicks, scale final amplitude, and output
adeclick lineseg 0.0, 0.05, 1.0, p3 - 0.1, 1.0, 0.05, 0.0
outs    ay * adeclick * 10000, ay * adeclick * 10000
endin

</CsInstruments>
<CsScore>
f1 0 32768 10 1 ; a sine wave

i1 0 5
e
```

```
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Author: Anthony Kozar  
January 2008

New in Csound version 5.08

# checkbox

checkbox — Sense on-screen controls.

## Description

Sense on-screen controls. Requires Winsound or TCL/TK.

## Syntax

```
kres checkbox knum
```

## Performance

Note that this opcode is not available on Windows due to the implimentation of pipes on that system

*kres* -- value of the checkbox control. If the checkbox is set (pushed) then return 1, if not, return 0.

*knun* -- the number of the checkbox. If it does not exist, it is made on-screen at initialization.

## Examples

Here is a simple example of the checkbox opcode. It uses the file *checkbox.csd* [examples/checkbox.csd].

### Example 120. Simple example of the checkbox opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o checkbox.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 10
nchnls = 2

instr 1
; Get the value from the checkbox.
k1 checkbox 1

; If the checkbox is selected then k2=440, otherwise k2=880.
k2 = (k1 == 0 ? 440 : 880)

a1 oscil 10000, k2, 1
outs a1, a1
kq button 1
if kq = 1 then
    exitnow
endif
```

```
    endin

    </CsInstruments>
    <CsScore>

    ; sine wave.
    f 1 0 32768 10 1

    i 1 0 1000
    e

    </CsScore>
</CsoundSynthesizer>
```

## See Also

*button*

## Credits

Author: John ffitch  
University of Bath, Codemist. Ltd.  
Bath, UK  
September, 2000

Example written by Kevin Conder.

New in Csound version 4.08

# chn

chn — Declare a channel of the named software bus.

## Description

Declare a channel of the named software bus, with setting optional parameters in the case of a control channel. If the channel does not exist yet, it is created, with an initial value of zero or empty string. Otherwise, the type (control, audio, or string) of the existing channel must match the declaration, or an init error occurs. The input/output mode of an existing channel is updated so that it becomes the bitwise OR of the previous and the newly specified value.

## Syntax

```
chn_k Sname, imode[, itype, idflt, imin, ima, ix, iy, iwidth, iheight, Sattributes]
```

```
chn_a Sname, imode
```

```
chn_s Sname, imode
```

## Initialization

*imode* -- sum of at least one of 1 for input and 2 for output.

*itype* (optional, defaults to 0) -- channel subtype for control channels only. Possible values are:

- 0: default/unspecified (*idflt*, *imin*, and *imax* are ignored)
- 1: integer values only
- 2: linear scale
- 3: exponential scale

*idflt* (optional, defaults to 0) -- default value, for control channels with non-zero *itype* only. Must be greater than or equal to *imin*, and less than or equal to *imax*.

*imin* (optional, defaults to 0) -- minimum value, for control channels with non-zero *itype* only. Must be non-zero for exponential scale (*itype* = 3).

*imax* (optional, defaults to 0) -- maximum value, for control channels with non-zero *itype* only. Must be greater than *imin*. In the case of exponential scale, it should also match the sign of *imin*.

*ix* -- suggested x position for controller.

*iy* -- suggested y position for controller.

*iwidth* -- suggested width position for controller.

*iheight* -- suggested height position for controller.

*Sattributes* -- attributes for controller.

## Notes

The channel parameters (*imode*, *itype*, *idflt*, *imin*, and *imax*) are only hints for the host application or external software accessing the bus through the API, and do not actually restrict reading from or writing to the channel in any way. Also, the initial value of a newly created control channel is zero, regardless of the setting of *idflt*.

For communication with external software, using *chnexport* may be preferred, as it allows direct access to orchestra variables exported as channels of the bus, eliminating the need for using *chnset* and *chnget* to send or receive data.

## Performance

**chn\_k**, **chn\_a**, and **chn\_S** declare a control, audio, or string channel, respectively.

## Example

The example shows the software bus being used as an asynchronous control signal to select a filter cutoff. It assumes that an external program that has access to the API is feeding the values.

```
sr = 44100
kr = 100
ksmps = 1

chn_k "cutoff", 1, 3, 1000, 500, 2000

instr 1
  kc chnget "cutoff"
  a1 oscil p4, p5, 100
  a2 lowpass2 a1, kc, 200
  out a2
endin
```

## Credits

Author: Istvan Varga  
2005

# chnclear

chnclear — Clears an audio output channel of the named software bus.

## Description

Clears an audio channel of the named software bus to zero. Implies declaring the channel with *imode=2* (see also *chn\_a*).

## Syntax

```
chnclear Sname
```

## Initialization

*Sname* -- a string that indicates which named channel of the software bus to clear.

## Examples

Here is an example of the chnclear opcode. It uses the file *chnclear.csd* [examples/chnclear.csd].

### Example 121. Example of the chnclear opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o chnclear.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1; send i-values
    chnset 1, "sio"
    chnset -1, "non"
endin

instr 2; send k-values
kfreq randomi 100, 300, 1
    chnset kfreq, "cntrfreq"
kbw = kfreq/10
    chnset kbw, "bandw"
endin

instr 3; send a-values
anois rand .1
    chnset anois, "noise"

loop:
idur random .3, 1.5
    timeout 0, idur, do
```

```

    reinit    loop
do:
ifreq    random    400, 1200
iamp     random    .1, .3
asig     oscils    iamp, ifreq, 0
aenv     transeg   1, idur, -10, 0
asine    =        asig * aenv
          chnset   asine, "sine"
        endin

        instr 11; receive some chn values and send again
ival1    chnget   "sio"
ival2    chnget   "non"
          print   ival1, ival2
kcntfreq chnget   "cntrfreq"
kbandw   chnget   "bandw"
anoise   chnget   "noise"
afilt    reson   anoise, kcntfreq, kbandw
afilt    balance afilt, anoise
          chnset   afilt, "filtered"
        endin

        instr 12; mix the two audio signals
amix1    chnget   "sine"
amix2    chnget   "filtered"
          chnmix   amix1, "mix"
          chnmix   amix2, "mix"
        endin

        instr 20; receive and reverb
amix     chnget   "mix"
aL, aR   freeverb amix, amix, .8, .5
          outs    aL, aR
        endin

        instr 100; clear
          chnclear "mix"
        endin

</CsInstruments>
<CsScore>
i 1 0 20
i 2 0 20
i 3 0 20
i 11 0 20
i 12 0 20
i 20 0 20
i 100 0 20
</CsScore>
</CsoundSynthesizer>

```

## Credits

Author: Istvan Varga  
2006

# chnexport

chnexport — Export a global variable as a channel of the bus.

## Description

Export a global variable as a channel of the bus; the channel should not already exist, otherwise an init error occurs. This opcode is normally called from the orchestra header, and allows the host application to read or write orchestra variables directly, without having to use *chnget* or *chnset* to copy data.

## Syntax

```
gival chnexport Sname, imode[, itype, idflt, imin, imax]
gkval chnexport Sname, imode[, itype, idflt, imin, imax]
gaval chnexport Sname, imode
gSval chnexport Sname, imode
```

## Initialization

*imode* -- sum of at least one of 1 for input and 2 for output.

*itype* (optional, defaults to 0) -- channel subtype for control channels only. Possible values are:

- 0: default/unspecified (*idflt*, *imin*, and *imax* are ignored)
- 1: integer values only
- 2: linear scale
- 3: exponential scale

*idflt* (optional, defaults to 0) -- default value, for control channels with non-zero *itype* only. Must be greater than or equal to *imin*, and less than or equal to *imax*.

*imin* (optional, defaults to 0) -- minimum value, for control channels with non-zero *itype* only. Must be non-zero for exponential scale (*itype* = 3).

*imax* (optional, defaults to 0) -- maximum value, for control channels with non-zero *itype* only. Must be greater than *imin*. In the case of exponential scale, it should also match the sign of *imin*.

## Notes

The channel parameters (*imode*, *itype*, *idflt*, *imin*, and *imax*) are only hints for the host application or external software accessing the bus through the API, and do not actually restrict reading from or writing to the channel in any way.

While the global variable is used as output argument, *chnexport* does not actually change it, and always runs at i-time only. If the variable is not previously declared, it is created by Csound with an initial value of zero or empty string.

## Example

The example shows the software bus being used as an asynchronous control signal to select a filter cutoff. It assumes that an external program that has access to the API is feeding the values.

```
sr = 44100
kr = 100
ksmps = 1

gkc init 1000 ; set default value
gkc chnexport "cutoff", 1, 3, i(gkc), 500, 2000

instr 1
  a1 oscil      p4, p5, 100
  a2 lowpass2  a1, gkc, 200
  out          a2
endin
```

## Credits

Author: Istvan Varga  
2005

# chnget

chnget — Reads data from the software bus.

## Description

Reads data from a channel of the inward named software bus. Implies declaring the channel with *imode=1* (see also *chn\_k*, *chn\_a*, and *chn\_S*).

## Syntax

```
ival chnget Sname  
kval chnget Sname  
aval chnget Sname  
Sval chnget Sname
```

## Initialization

*Sname* -- a string that identifies a channel of the named software bus to read.

*ival* -- the control value read at i-time.

*Sval* -- the string value read at i-time.

## Performance

*kval* -- the control value read at performance time.

*aval* -- the audio signal read at performance time.

## Example

The example shows the software bus being used as an asynchronous control signal to select a filter cutoff. It assumes that an external program that has access to the API is feeding the values.

```
sr = 44100  
kr = 100  
ksmps = 1  
  
instr 1  
  kc  chnget  "cutoff"  
  a1  oscil   p4, p5, 100  
  a2  lowpass2 a1, kc, 200  
  out  a2  
endin
```

Here is another example of the chnget opcode. It uses the file *chnget.csd* [examples/chnget.csd].

**Example 122. Example of the chnget opcode.**

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o chnget.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1; send i-values
    chnset 1, "sio"
    chnset -1, "non"
endin

instr 2; send k-values
kfreq random 100, 300, 1
    chnset kfreq, "cntrfreq"
kbw = kfreq/10
    chnset kbw, "bandw"
endin

instr 3; send a-values
anois rand .1
    chnset anois, "noise"

loop:
idur random .3, 1.5
    timeout 0, idur, do
    reinit loop

do:
ifreq random 400, 1200
iamp random .1, .3
asig oscils iamp, ifreq, 0
aenv transeg 1, idur, -10, 0
asine = asig * aenv
    chnset asine, "sine"
endin

instr 11; receive some chn values and send again
ival1 chnget "sio"
ival2 chnget "non"
print ival1, ival2
kcntfreq chnget "cntrfreq"
kbandw chnget "bandw"
anoise chnget "noise"
afilt reson anoise, kcntfreq, kbandw
afilt balance afilt, anoise
    chnset afilt, "filtered"
endin

instr 12; mix the two audio signals
amix1 chnget "sine"
amix2 chnget "filtered"
    chnmix amix1, "mix"
    chnmix amix2, "mix"
endin

instr 20; receive and reverb

```

```
amix      chnget      "mix"  
aL, aR   freeverb   amix, amix, .8, .5  
         outs       aL, aR  
  
    endin  
  
    instr 100; clear  
         chnclear  "mix"  
    endin  
  
</CsInstruments>  
<CsScore>  
i 1 0 20  
i 2 0 20  
i 3 0 20  
i 11 0 20  
i 12 0 20  
i 20 0 20  
i 100 0 20  
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Author: Istvan Varga  
2005

# chnmix

chnmix — Writes audio data to the named software bus, mixing to the previous output.

## Description

Adds an audio signal to a channel of the named software bus. Implies declaring the channel with *imode=2* (see also *chn\_a*).

## Syntax

```
chnmix aval, Sname
```

## Initialization

*Sname* -- a string that indicates which named channel of the software bus to write.

## Performance

*aval* -- the audio signal to write at performance time.

## Examples

Here is an example of the chnmix opcode. It uses the file *chnmix.csd* [examples/chnmix.csd].

### Example 123. Example of the chnmix opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o chnmix.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1; send i-values
  chnset 1, "sio"
  chnset -1, "non"
endin

instr 2; send k-values
kfreq  randomi 100, 300, 1
  chnset kfreq, "cntrfreq"
kbw    =      kfreq/10
  chnset kbw, "bandw"
endin
```

```

    instr 3; send a-values
anois    rand    .1
         chnset  anois, "noise"

loop:
idur     random  .3, 1.5
         timeout 0, idur, do
         reinit  loop

do:
ifreq    random  400, 1200
iamp     random  .1, .3
asig     oscils  iamp, ifreq, 0
aenv     transeg 1, idur, -10, 0
asine    =       asig * aenv
         chnset  asine, "sine"
    endin

    instr 11; receive some chn values and send again
ival1    chnget  "sio"
ival2    chnget  "non"
         print  ival1, ival2
kcntfreq chnget  "cntrfreq"
kbandw   chnget  "bandw"
anoise    chnget  "noise"
afilt     reson  anoise, kcntfreq, kbandw
afilt     balance afilt, anoise
         chnset  afilt, "filtered"
    endin

    instr 12; mix the two audio signals
amix1     chnget  "sine"
amix2     chnget  "filtered"
         chnmix  amix1, "mix"
         chnmix  amix2, "mix"
    endin

    instr 20; receive and reverb
amix      chnget  "mix"
aL, aR    freeverb amix, amix, .8, .5
         outs   aL, aR
    endin

    instr 100; clear
         chnclear "mix"
    endin

</CsInstruments>
<CsScore>
i 1 0 20
i 2 0 20
i 3 0 20
i 11 0 20
i 12 0 20
i 20 0 20
i 100 0 20
</CsScore>
</CsSoundSynthesizer>

```

## Credits

Author: Istvan Varga  
2006

# chnparams

chnparams — Query parameters of a channel.

## Description

Query parameters of a channel (if it does not exist, all returned values are zero).

## Syntax

```
itype, imode, ictltype, idflt, imin, imax chnparams Sname
```

## Initialization

*itype* -- channel data type (1: control, 2: audio, 3: string)

*imode* -- sum of 1 for input and 2 for output

*ictltype* -- special parameter for control channel only; if not available, set to zero.

*idflt* -- special parameter for control channel only; if not available, set to zero.

*imin* -- special parameter for control channel only; if not available, set to zero.

*imax* -- special parameter for control channel only; if not available, set to zero.

*Sname* -- string identifying the channel.

## Credits

Author: Istvan Varga  
2005

# chnset

chnset — Writes data to the named software bus.

## Description

Write to a channel of the named software bus. Implies declaring the channel with *imod=2* (see also *chn\_k*, *chn\_a*, and *chn\_S*).

## Syntax

```
chnset ival, Sname
chnset kval, Sname
chnset aval, Sname
chnset sval, Sname
```

## Initialization

*Sname* -- a string that indicates which named channel of the software bus to write.

*ival* -- the control value to write at i-time.

*Sval* -- the string value to write at i-time.

## Performance

*kval* -- the control value to write at performance time.

*aval* -- the audio signal to write at performance time.

## Example

The example shows the software bus being used to write pitch information to a controlling program.

```
sr = 44100
kr = 100
ksmps = 1

instr 1
  al in
  kp,ka pitchamdf al
  chnset kp, "pitch"
endin
```

Here is another example of the chnset opcode. It uses the file *chnset.csd* [examples/chnset.csd].

### Example 124. Example of the chnset opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
```

```

; For Non-realtime ouput leave only the line below:
; -o chnset.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1; send i-values
    chnset 1, "sio"
    chnset -1, "non"
endin

instr 2; send k-values
kfreq randomi 100, 300, 1
    chnset kfreq, "cntrfreq"
kbw = kfreq/10
    chnset kbw, "bandw"
endin

instr 3; send a-values
anois rand .1
    chnset anois, "noise"

loop:
idur random .3, 1.5
    timeout 0, idur, do
    reinit loop

do:
ifreq random 400, 1200
iamp random .1, .3
asig oscils iamp, ifreq, 0
aenv transeg 1, idur, -10, 0
asine = asig * aenv
    chnset asine, "sine"
endin

instr 11; receive some chn values and send again
ival1 chnget "sio"
ival2 chnget "non"
print ival1, ival2
kcntfreq chnget "cntrfreq"
kbandw chnget "bandw"
anoise chnget "noise"
afilt reson anoise, kcntfreq, kbandw
afilt balance afilt, anoise
    chnset afilt, "filtered"
endin

instr 12; mix the two audio signals
amix1 chnget "sine"
amix2 chnget "filtered"
    chnmix amix1, "mix"
    chnmix amix2, "mix"
endin

instr 20; receive and reverb
amix chnget "mix"
aL, aR freeverb amix, amix, .8, .5
    outs aL, aR
endin

instr 100; clear
    chnclear "mix"
endin

```

```
</CsInstruments>  
<CsScore>  
i 1 0 20  
i 2 0 20  
i 3 0 20  
i 11 0 20  
i 12 0 20  
i 20 0 20  
i 100 0 20  
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Author: Istvan Varga  
2005

# chuap

chuap — Simulates Chua's oscillator, an LRC oscillator with an active resistor, proved capable of bifurcation and chaotic attractors, with k-rate control of circuit elements.

## Description

Simulates Chua's oscillator, an LRC oscillator with an active resistor, proved capable of bifurcation and chaotic attractors, with k-rate control of circuit elements.

## Syntax

```
aI3, aV2, aV1 chuap kL, kR0, kC1, kG, kGa, kGb, kE, kC2, iI3, iV2, iV1, ktime_step
```

## Initialization

*iI3* -- Initial current at G

*iV2* -- Initial voltage at C2

*iV1* -- Initial voltage at C1

## Performance

*kL* -- Inductor L

*kR0* -- Resistor R0

*kC1* -- Capacitor C1

*kG* -- Resistor G

*kGa* -- Resistor V (nonlinearity term)

*kGb* -- Resistor V (nonlinearity term)

*kGb* -- Resistor V (nonlinearity term)

*ktime\_step* -- Delta time in the difference equation, can be used to more or less control pitch.

*Chua's oscillator* is a simple LRC oscillator with an active resistor. The oscillator can be driven into period bifurcation, and thus to chaos, because of the nonlinear response of the active resistor.

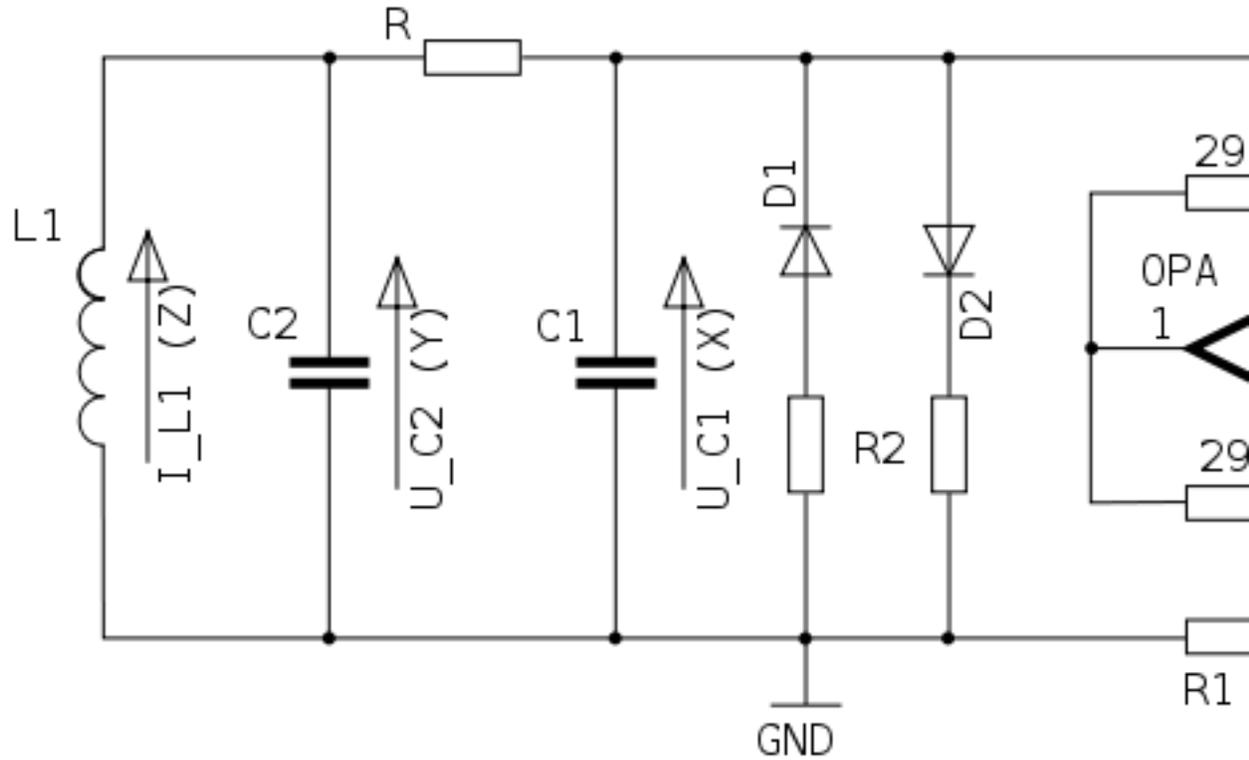


Diagram of Chua's Oscillator Circuit

The circuit is described by a set of three ordinary differential equations called Chua's equations:

$$\frac{dI_3}{dt} = -\frac{R_0}{L} I_3 - \frac{1}{L} V_2$$

$$\frac{dV_2}{dt} = -\frac{1}{C_2} I_3 - \frac{G}{C_2} (V_2 - V_1)$$

$$\frac{dV_1}{dt} = -\frac{G}{C_1} (V_2 - V_1) - \frac{1}{C_1} f(V_1)$$

where  $f()$  is a piecewise discontinuity simulating the active resistor:

$$f(V_1) = G_b V_1 + -(G_a - G_b)(|V_1 + E| - |V_1 - E|)$$

A solution of these equations  $(I_3, V_2, V_1)(t)$  starting from an initial state  $(I_3, V_2, V_1)(0)$  is called a trajectory of Chua's oscillator. The Csound implementation is a difference equation simulation of Chua's oscillator with Runge-Kutta integration.



## Note

This algorithm uses internal non linear feedback loops which causes audio result to depend on the orchestra sampling rate. For example, if you develop a project with  $sr=48000\text{Hz}$  and if you want to produce an audio CD from it, you should record a file with  $sr=48000\text{Hz}$  and then downsample the file to  $44100\text{Hz}$  using the *srconv* utility.



## Warning

Be careful! Some sets of parameters will produce amplitude spikes or positive feedback that could damage your speakers.

## Examples

Here is an example of the *chuap* opcode. It uses the file *chuap.csd* [examples/chuap.csd].

### Example 125. Example of the *chuap* opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o chuas_oscillator.wav.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr      = 44100
ksmps  = 32
nchnls = 2
0dbfs  = 1

gibuzztable ftgen 1, 0, 16384, 10, 1

instr 1
; sys_variables = system_vars(5:12); % L,R0,C2,G,Ga,Gb,E,C1 or p8:p15
; integ_variables = [system_vars(14:16),system_vars(1:2)]; % x0,y0,z0,dataset_size,step_size or p17:p19
istep_size = p5
iL         = p8
iR0        = p9
iC2        = p10
iG         = p11
iGa        = p12
iGb        = p13
iE         = p14
iC1        = p15
iI3        = p17
iV2        = p18
iV1        = p19
ignore     = p21
ignore     = p22
ignore     = p23
ignore     = p24
ignore     = p25
iattack    = 0.02
isustain   = p3
irelease   = 0.02
p3         = iattack + isustain + irelease
iscale     = 1.0
adamping   linseg 0.0, iattack, iscale, isustain, iscale, irelease, 0.0
aguide     buzz 0.5, 440, sr/440, gibuzztable
```

```
aI3, aV2, aV1 chuap iL, iR0, iC2, iG, iGa, iGb, iE, iC1, iI3, iV2, iV1, istep_size
asignal balance aV2, aguide
outs adamping * asignal, adamping * asignal

endin
</CsInstruments>
<CsScore>
; Adapted from ABC++ MATLAB example data.
i 1 0 20 1500 .1 -1 -1 -0.00707925 0.00001647 100 1 -.99955324 -1.00028375 1 -.00222159 204.8 -2.362
i 1 + 20 1500 .425 0 -1 1.3506168 0 -4.50746268737 -1 2.4924 .93 1 1 0 -22.28662665 .00
i 1 + 20 1024 .05 -1 -1 0.00667 0.000651 10 -1 .856 1.1 1 .06 51.2 -20.200590133667 .172539323
i 1 + 20 1024 0.05 -1 -1 0.00667 0.000651 10 -1 0.856 1.1 1 0.1 153.6 21.12496758 0.03001749 0.5158286
</CsScore>
</CsoundSynthesizer>
```

## Credits

Inventor of Chua's oscillator: *Leon O. Chua* [<http://www.eecs.berkeley.edu/~chua>]

Author of MATLAB simulation: James Patrick McEvoy *MATLAB Adventures in Bifurcations and Chaos* (ABC++) [<http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=3541>]

Author of Csound port: Michael Gogins

New in Csound version 5.09

Note added by François Pinot, August 2009

# cigoto

cigoto — Conditionally transfer control during the i-time pass.

## Description

During the i-time pass only, conditionally transfer control to the statement labeled by *label*.

## Syntax

```
cigoto condition, label
```

where *label* is in the same instrument block and is not an expression, and where *condition* uses one of the Relational operators (<, =, <=, ==, !=) (and = for convenience, see also under *Conditional Values*).

## Examples

Here is an example of the cigoto opcode. It uses the file *cigoto.csd* [examples/cigoto.csd].

### Example 126. Example of the cigoto opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o cigoto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Get the value of the 4th p-field from the score.
iparam = p4

; If iparam is 1 then play the high note.
; If not then play the low note.
cigoto (iparam ==1), highnote
      igoto lownote

highnote:
  ifreq = 880
  goto playit

lownote:
  ifreq = 440
  goto playit
```

```
playit:
  ; Print the values of iparam and ifreq.
  print iparam
  print ifreq

  a1 oscil 10000, ifreq, 1
  out a1
endin

</CsInstruments>
<CsScore>

; Table #1: a simple sine wave.
f 1 0 32768 10 1

; p4: 1 = high note, anything else = low note
; Play Instrument #1 for one second, a low note.
i 1 0 1 0
; Play a Instrument #1 for one second, a high note.
i 1 1 1 1
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like:

```
instr 1: iparam = 0.000
instr 1: ifreq = 440.000
instr 1: iparam = 1.000
instr 1: ifreq = 880.000
```

## See Also

*cggoto, ckgoto, cngoto, goto, if, kgoto, rigoto, tigoto, timeout*

## Credits

Example written by Kevin Conder.

# ckgoto

ckgoto — Conditionally transfer control during the p-time passes.

## Description

During the p-time passes only, conditionally transfer control to the statement labeled by *label*.

## Syntax

```
ckgoto condition, label
```

where *label* is in the same instrument block and is not an expression, and where *condition* uses one of the Relational operators (<, =, <=, ==, !=) (and = for convenience, see also under *Conditional Values*).

## Examples

Here is an example of the ckgoto opcode. It uses the file *ckgoto.csd* [examples/ckgoto.csd].

### Example 127. Example of the ckgoto opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o ckgoto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Change kval linearly from 0 to 2 over
; the period set by the third p-field.
kval line 0, p3, 2

; If kval is greater than or equal to 1 then play the high note.
; If not then play the low note.
ckgoto (kval >= 1), highnote
      kgoto lownote

highnote:
      kfreq = 880
      goto playit

lownote:
      kfreq = 440
```

```
    goto playit

playit:
    ; Print the values of kval and kfreq.
    printks "kval = %f, kfreq = %f\\n", 1, kval, kfreq

    a1 oscil 10000, kfreq, 1
    out a1
endin

</CsInstruments>
<CsScore>

    ; Table: a simple sine wave.
    f 1 0 32768 10 1

    ; Play Instrument #1 for two seconds.
    i 1 0 2
    e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like:

```
kval = 0.000000, kfreq = 440.000000
kval = 0.999732, kfreq = 440.000000
kval = 1.999639, kfreq = 880.000000
```

## See Also

*cggoto, cigoto, cngoto, goto, if, igoto, tigoto, timeout*

## Credits

Example written by Kevin Conder.

# clear

clear — Zeroes a list of audio signals.

## Description

*clear* zeroes a list of audio signals.

## Syntax

```
clear avar1 [, avar2] [, avar3] [...]
```

## Performance

*avar1*, *avar2*, *avar3*, ... -- signals to be zeroed

*clear* sets every sample of each of the given audio signals to zero when it is performed. This is equivalent to writing  $avarN = 0$  in the orchestra for each of the specified variables. Typically, *clear* is used with global variables that combine multiple signals from different sources and change with each k-pass (performance loop) through all of the active instrument instances. After the final usage of such a variable and before the next k-pass, it is necessary to clear the variable so that it does not add the next cycle's signals to the previous result. *clear* is especially useful in combination with *vincr* (variable increment) and they are intended to be used together with file output opcodes such as *fout*.

## Examples

Here is an example of the clear opcode. It uses the file *clear.csd* [examples/clear.csd].

### Example 128. Example of the clear opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o clear.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gaReverb init 0

instr 1

idur = p3
kpitch = p4
a1 diskin2 "fox.wav", kpitch
a1 = a1*.5 ;reduce volume
vincr gaReverb, a1
```

```
    endin

    instr 99 ; global reverb
    al, ar reverbsc gaReverb, gaReverb, .8, 10000
    outs gaReverb+al, gaReverb+ar

    clear gaReverb

    endin

    </CsInstruments>
    <CsScore>

    i1 0 3 1
    i99 0 5
    e

    </CsScore>
    </CsoundSynthesizer>
```

See the *fout* opcode for another example.

## See Also

*vincr*

## Credits

Author: Gabriel Maldonado  
Italy  
1999

New in Csound version 3.56

# clfilt

clfilt — Implements low-pass and high-pass filters of different styles.

## Description

Implements the classical standard analog filter types: low-pass and high-pass. They are implemented with the four classical kinds of filters: Butterworth, Chebyshev Type I, Chebyshev Type II, and Elliptical. The number of poles may be any even number from 2 to 80.

## Syntax

```
ares clfilt asig, kfreq, itype, inpol [, ikind] [, ipbr] [, isba] [, iskip]
```

## Initialization

*itype* -- 0 for low-pass, 1 for high-pass.

*inpol* -- The number of poles in the filter. It must be an even number from 2 to 80.

*ikind* (optional) -- 0 for Butterworth, 1 for Chebyshev Type I, 2 for Chebyshev Type II, 3 for Elliptical. Defaults to 0 (Butterworth)

*ipbr* (optional) -- The pass-band ripple in dB. Must be greater than 0. It is ignored by Butterworth and Chebyshev Type II. The default is 1 dB.

*isba* (optional) -- The stop-band attenuation in dB. Must be less than 0. It is ignored by Butterworth and Chebyshev Type I. The default is -60 dB.

*iskip* (optional) -- 0 initializes all filter internal states to 0. 1 skips initialization. The default is 0.

## Performance

*asig* -- The input audio signal.

*kfreq* -- The corner frequency for low-pass or high-pass.

## Examples

Here is an example of the clfilt opcode as a low-pass filter. It uses the file *clfilt\_lowpass.csd* [examples/clfilt\_lowpass.csd].

### Example 129. Example of the clfilt opcode as a low-pass filter.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
-odac      ;;RT audio out
```

```

;-iadc    ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o clfilt_lowpass.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; white noise

asig rand 0.5
outs asig, asig

endin

instr 2 ; filtered noise

asig rand 0.9
; Lowpass filter signal asig with a
; 10-pole Butterworth at 500 Hz.
a1 clfilt asig, 500, 0, 10
outs a1, a1

endin

</CsInstruments>
<CsScore>

i 1 0 2
i 2 2 2
e

</CsScore>
</CsoundSynthesizer>

```

Here is an example of the `clfilt` opcode as a high-pass filter. It uses the file `clfilt_highpass.csd` [examples/clfilt\_highpass.csd].

### Example 130. Example of the `clfilt` opcode as a high-pass filter.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;RT audio out
;-iadc    ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o clfilt_highpass.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; white noise

asig rand 0.6
outs asig, asig

```

```
endin

instr 2 ;filtered noise

asig rand 0.7
; Highpass filter signal asig with a 6-pole Chebyshev
; Type I at 20 Hz with 3 dB of passband ripple.
a1 clfilt asig, 20, 1, 6, 1, 3
outs a1, a1

endin

</CsInstruments>
<CsScore>

i 1 0 2
i 2 2 2
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Erik Spjut

New in version 4.20

# clip

`clip` — Clips a signal to a predefined limit.

## Description

Clips an a-rate signal to a predefined limit, in a “soft” manner, using one of three methods.

## Syntax

```
ares clip asig, imeth, ilimit [, iarg]
```

## Initialization

*imeth* -- selects the clipping method. The default is 0. The methods are:

- 0 = Bram de Jong method (default)
- 1 = sine clipping
- 2 = tanh clipping

*ilimit* -- limiting value

*iarg* (optional, default=0.5) -- when *imeth* = 0, indicates the point at which clipping starts, in the range 0 - 1. Not used when *imeth* = 1 or *imeth* = 2. Default is 0.5.

## Performance

*asig* -- a-rate input signal

The Bram de Jong method (*imeth* = 0) applies the algorithm (denoting *ilimit* as *limit* and *iarg* as *a*):

$$\begin{aligned} |x| \geq 0 \text{ and } |x| \leq (limit * a): & f(x) = f(x) \\ |x| > (limit * a) \text{ and } |x| \leq limit: & f(x) = \text{sign}(x) * (limit * a + (x - limit * a) / (1 + ((x - limit * a) / (limit * (1 - a)))) \\ |x| > limit: & f(x) = \text{sign}(x) * (limit * (1 + a)) / 2 \end{aligned}$$

The second method (*imeth* = 1) is the sine clip:

$$|x| < limit: f(x) = limit * \sin(\#x / (2 * limit)), \quad |x| \geq limit: f(x) = limit * \text{sign}(x)$$

The third method (*imeth* = 2) is the tanh clip:

$$|x| < limit: f(x) = limit * \tanh(x / limit) / \tanh(1), \quad |x| \geq limit: f(x) = limit * \text{sign}(x)$$


### Note

Method 1 appears to be non-functional at release of Csound version 4.07.

## Examples

Here is an example of the clip opcode. It uses the file *clip.csd* [examples/clip.csd].

### Example 131. Example of the clip opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o clip.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; white noise

arnd rand 1 ; full amlitude
; Clip the noisy waveform's amplitude to 0.5
a1 clip arnd, 2, 0.5
  outs a1, a1

endin

instr 2 ; white noise

arnd rand 1 ; full amlitude
; Clip the noisy waveform's amplitude to 0.1
a1 clip arnd, 2, 0.1
  outs a1, a1

endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #2 for one second.
i 2 1 1
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitth  
University of Bath, Codemist Ltd.  
Bath, UK  
August, 2000

New in Csound version 4.07

September 2009: Thanks to a note from Paolo Dell'Osso, corrected the formula.

# clockoff

clockoff — Stops one of a number of internal clocks.

## Description

Stops one of a number of internal clocks.

## Syntax

```
clockoff inum
```

## Initialization

*inum* -- the number of a clock. There are 32 clocks numbered 0 through 31. All other values are mapped to clock number 32.

## Performance

Between a *clockon* and a *clockoff* opcode, the CPU time used is accumulated in the clock. The precision is machine dependent but is the millisecond range on UNIX and Windows systems. The *readclock* opcode reads the current value of a clock at initialization time.

## Examples

Here is an example of the clockoff opcode. It uses the file *clockoff.csd* [examples/clockoff.csd].

### Example 132. Example of the clockoff opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o clockoff.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1.
instr 1
; Start clock #1.
clockon 1
; Do something that keeps Csound busy.
a1 oscili 10000, 440, 1
out a1
; Stop clock #1.
clockoff 1
```

```
    ; Print the time accumulated in clock #1.
    i1 readclock 1
    print i1
endin

</CsInstruments>
<CsScore>

; Initialize the function tables.
; Table 1: an ordinary sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for one second starting at 0:00.
i 1 0 1
; Play Instrument #1 for one second starting at 0:01.
i 1 1 1
; Play Instrument #1 for one second starting at 0:02.
i 1 2 1
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*clockon, readclock*

## Credits

Author: John ffitth  
University of Bath/Codemist Ltd.  
Bath, UK  
July, 1999

New in Csound version 3.56

# clockon

clockon — Starts one of a number of internal clocks.

## Description

Starts one of a number of internal clocks.

## Syntax

```
clockon inum
```

## Initialization

*inum* -- the number of a clock. There are 32 clocks numbered 0 through 31. All other values are mapped to clock number 32.

## Performance

Between a *clockon* and a *clockoff* opcode, the CPU time used is accumulated in the clock. The precision is machine dependent but is the millisecond range on UNIX and Windows systems. The *readclock* opcode reads the current value of a clock at initialization time.

## Examples

Here is an example of the *clockon* opcode. It uses the file *clockon.csd* [examples/clockon.csd].

### Example 133. Example of the *clockon* opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o clockon.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1.
instr 1
; Start clock #1.
clockon 1
; Do something that keeps Csound busy.
a1 oscili 10000, 440, 1
out a1
; Stop clock #1.
clockoff 1
```

```
    ; Print the time accumulated in clock #1.
    i1 readclock 1
    print i1
endin

</CsInstruments>
<CsScore>

; Initialize the function tables.
; Table 1: an ordinary sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for one second starting at 0:00.
i 1 0 1
; Play Instrument #1 for one second starting at 0:01.
i 1 1 1
; Play Instrument #1 for one second starting at 0:02.
i 1 2 1
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*clockoff, readclock*

## Credits

Author: John ffitth  
University of Bath/Codemist Ltd.  
Bath, UK  
July, 1999

New in Csound version 3.56

# cmp

cmp — Compares two audio signals

## Description

Compares two audio signals using the standard math operators

```
aout cmp aL, ">", aR
```

## Syntax

```
aout cmp aL, S_operator, aR
```

## Initialization

*S\_operator* Math operator, one of ">", ">=", "<", "<=", "=="

## Performance

*aL* -- Left signal

*aR* -- Right signal

*kdur* -- Duration of the trigger in seconds.

```
aout cmp aL, ">", aR      ; aout = aL > aR for every sample
aout cmp aL, ">=", aR
aout cmp aL, "<", aR
aout cmp aL, "<=", aR
aout cmp aL, "==", aR
```

## Examples

Here is an example of the cmp opcode. It uses the file *cmp.csd* [examples/cmp.csd].

### Example 134. Example of the cmp opcode.

```
<CsoundSynthesizer>
<CsoundOptions>
</CsoundOptions>
<CsoundInstruments>

sr = 44100
ksmps = 128
nchnls = 6
0dbfs = 1.0

instr 1
```

```
    a0 linseg 0, p3, 1
    a1 linseg 1, p3, 0
    aout cmp a0, "<", a1
    outch 1, a0
  outch 2, a1
    outch 3, aout
  endin

instr 2
  a0 linseg 0, p3, 1
  aout cmp a0, "<=", 0.5
  outch 4, aout
  endin

instr 3
  a0 linseg 0, p3, 1
  aout cmp a0, "!=", 0.5
  outch 5, aout
  endin

</CsInstruments>
<CsScore>
i 1 0 2
i 2 0 2
i 3 0 2

</CsScore>
</CsoundSynthesizer>
```

## See Also

*max, min,*

## Credits

By: Eduardo Moguillansky 2017

# cmplxprod

cmplxprod — Complex product of two arrays.

## Description

Calculates the complex product of two arrays of the same size and in real-imaginary interleaved format.

## Syntax

```
kout[] cmplxprod kin1[], kin2[]
```

## Performance

*kout[]* -- output array containing the product. It will be created if it does not exist.

*kin1[], kin2[]* -- input arrays containing the complex inputs.

## Examples

Here is an example of the cmplxprod opcode. It uses the file *cmplxprod.csd* [examples/cmplxprod.csd].

### Example 135. Example of the cmplxprod opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>

<CsOptions>
-d -o dac
</CsOptions>

<CsInstruments>
;ksmps needs to be an integer div of hopsize
ksmps = 64
0dbfs = 1

instr 1

  ihopsize = 256 ; hopsize
  ifftsize = 1024 ; FFT size
  iolaps = ifftsize/ihopsize ; overlaps
  ibw = sr/ifftsize ; bin bandwidth
  kent init 0 ; counting vars
  krow init 0

  kOla[] init ifftsize ; overlap-add buffer
  kIn[] init ifftsize ; input buffer
  kFil[] init ifftsize ; filter buffer
  kOut[][] init iolaps, ifftsize ; output buffers

  kfrst init 1
  if kfrst == 1 then
    copyf2array kFil,1
```

```

    kfst = 0
endif

a1 diskin2 "fox.wav",1,0,1 ; audio input

/* every hopsize samples */
if kcnt == ihopsize then
    /* window and take FFT */
    kWin[] window kIn,krow*ihopsize
    kSpec[] rfft kWin

    kProd[] cmplxprod kSpec, kFil

    /* IFFT + window */
    kRow[] rifft kProd
    kWin window kRow, krow*ihopsize
    /* place it on out buffer */
    kOut setrow kWin, krow

    /* zero the ola buffer */
    kOla = 0
    /* overlap-add */
    ki = 0
    until ki == iolaps do
        kRow getrow kOut, ki
        kOla = kOla + kRow
        ki += 1
    od

    /* update counters */
    krow = (krow+1)%iolaps
    kcnt = 0
endif

/* shift audio in/out of buffers */
kIn shiftin a1
a2 shiftout kOla
    out a2/iolaps

/* increment counter */
kcnt += ksmps

endin

</CsInstruments>

<CsScore>
f1 0 1024 7 0 64 0.1 64 0.2 128 0.5 256 1 512 1
i1 0 10
</CsScore>

</CsoundSynthesizer>

```

## See Also

Vectorial opcodes, array opcodes

## Credits

Author: Victor Lazzarini  
 NUI Maynooth  
 2014

New in version 6.04

# cngoto

cngoto — Transfers control on every pass when a condition is not true.

## Description

Transfers control on every pass when the condition is *not* true.

## Syntax

```
cngoto condition, label
```

where *label* is in the same instrument block and is not an expression, and where *condition* uses one of the Relational operators (<, =, <=, ==, !=) (and = for convenience, see also under *Conditional Values*).

## Examples

Here is an example of the cngoto opcode. It uses the file *cngoto.csd* [examples/cngoto.csd].

### Example 136. Example of the cngoto opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Audio out   Audio in
-odac         -iadc   ;;RT audio I/O
; -o cngoto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Change kval linearly from 0 to 2 over
; the period set by the third p-field.
kval line 0, p3, 2

; If kval *is not* greater than or equal to 1 then play
; the high note. Otherwise, play the low note.
cngoto (kval >= 1), highnote
      kgoto lownote

highnote:
      kfreq = 880
      goto playit

lownote:
      kfreq = 440
      goto playit

playit:
```

```
; Print the values of kval and kfreq.
printks "kval = %f, kfreq = %f\\n", 1, kval, kfreq

a1 oscil 10000, kfreq, 1
out a1
endin

</CsInstruments>
<CsScore>

; Table: a simple sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like:

```
kval = 0.000000, kfreq = 880.000000
kval = 0.999732, kfreq = 880.000000
kval = 1.999639, kfreq = 440.000000
```

## See Also

*eggoto, cigoto, ckgoto, goto, if, igoto, tigoto, timeout*

## Credits

Example written by Kevin Conder.

New in version 4.21

# comb

comb — Reverberates an input signal with a “colored” frequency response.

## Description

Reverberates an input signal with a “colored” frequency response.

## Syntax

```
ares comb asig, krvt, ilpt [, iskip] [, insmps]
```

## Initialization

*ilpt* -- loop time in seconds, which determines the “echo density” of the reverberation. This in turn characterizes the “color” of the *comb* filter whose frequency response curve will contain  $ilpt * sr/2$  peaks spaced evenly between 0 and  $sr/2$  (the Nyquist frequency). Loop time can be as large as available memory will permit. The space required for an  $n$  second loop is  $n*sr$  floating or double numbers (usually 4 or 8 bytes). Delay space is allocated and returned as in *delay*.

*iskip* (optional, default=0) -- initial disposition of delay-loop data space (cf. *reson*). The default value is 0.

*insmps* (optional, default=0) -- delay amount, as a number of samples.

## Performance

*krvt* -- the reverberation time (defined as the time in seconds for a signal to decay to 1/1000, or 60dB down from its original amplitude).

This filter reiterates input with an echo density determined by loop time *ilpt*. The attenuation rate is independent and is determined by *krvt*, the reverberation time (defined as the time in seconds for a signal to decay to 1/1000, or 60dB down from its original amplitude). Output from a comb filter will appear only after *ilpt* seconds.

## Examples

Here is an example of the comb opcode. It uses the file *comb.csd* [examples/comb.csd].

### Example 137. Example of the comb opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o comb.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```
sr = 44100
ksmps = 32
nchnls = 2
odbfs = 1

gamix init 0

instr 1

kcps    expon p5, p3, p4
asig vco2 0.3, kcps
outs asig, asig

gamix = gamix + asig

endin

instr 99

krvt = 3.5
ilpt = 0.1
aleft comb gamix, krvt, ilpt
aright comb gamix, krvt, ilpt*.2
outs  aleft, aright

clear gamix ; clear mixer

endin

</CsInstruments>
<CsScore>

i 1 0 3 20 2000
i 1 5 .01 440 440

i 99 0 8
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*alpass, reverb, valpass, vcomb*

## Credits

Barry L. Vercoe

# combinv

combinv — Reverberates an input signal with a “colored” frequency response.

## Description

Reverberates an input signal with a “colored” frequency response with a FIR filter.

## Syntax

```
ares combinv asig, krvt, ilpt [, iskip] [, insmps]
```

## Initialization

*ilpt* -- loop time in seconds, which determines the “echo density” of the reverberation. This in turn characterizes the “color” of the *combinv* filter whose frequency response curve will contain *ilpt* \* *sr*/2 peaks spaced evenly between 0 and *sr*/2 (the Nyquist frequency). Loop time can be as large as available memory will permit. The space required for an *n* second loop is *n*\**sr* floating or double numbers (usually 4 or 8 bytes). Delay space is allocated and returned as in *delay*.

*iskip* (optional, default=0) -- initial disposition of delay-loop data space (cf. *reson*). The default value is 0.

*insmps* (optional, default=0) -- delay amount, as a number of samples.

## Performance

*krvt* -- the reverberation time (defined as the time in seconds for a signal to decay to 1/1000, or 60dB down from its original amplitude).

This filter reiterates input with an echo density determined by loop time *ilpt*. The attenuation rate is independent and is determined by *krvt*, the reverberation time (defined as the time in seconds for a signal to decay to 1/1000, or 60dB down from its original amplitude). Affect from a *combinv* filter will appear only after *ilpt* seconds.

## Examples

Here is an example of the *combinv* opcode. It uses the file *combinv.csd* [examples/combinv.csd].

### Example 138. Example of the *combinv* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o comb.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kcps    expon p5, p3, p4
asig    oscil3 0.3, kcps
krvt = 3.5
ilpt = 0.1
aleft    combinv asig, krvt, ilpt
outs    aleft, asig

endin

</CsInstruments>
<CsScore>
i 1 0 3 20 2000
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*alpass, comb reverb, valpass, vcomb*

## Credits

Author: John ffitch  
October 2012

# compilecsd

compilecsd — compiles a new orchestra from an ASCII file

## Description

Compilecsd will read a CSD file and compile one or more instruments at init time, which will be added to the running engine. In case of existing instrument numbers or names, these will be replaced, but any instance still running of the old instrument definition will still perform until it terminates. In addition, it will read the score (if it exists) contained in the CSD file and add it to the list of events to be performed by Csound. The opcode ignores any section in the CSD file that is not the orchestra or the score.

## Syntax

```
ires compilecsd Sfilename
```

## Initialization

“*Sfilename*” -- a string containing the name of the file containing the orchestra.

“*ires*” -- returns 0 if compilation was successful, or -1 if not.

## Examples

Here is an example of the compilecsd opcode. It uses the file *compilecsd.csd* [examples/compilecsd.csd].

### Example 139. Example of the compilecsd opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-o dac -d
</CsOptions>
<CsInstruments>
sr = 44100
nchnls = 1
ksmps = 32
Odbfs = 1

instr 1
ires compilecsd "does_not_exist.csd"
print ires ; -1 as could not compile
ires compilecsd "my.csd"
print ires ; 0 as compiled successfully
event_i "i", 2, 0, 3, .2, 465 ;send event
endin

</CsInstruments>
<CsScore>
i1 0 1
</CsScore>
</CsoundSynthesizer>
```

## See also

*event, event\_i, schedule, schedwhen, schedkwhen, schedkwhennamed*

## Credits

Author: Victor Lazzarini, 2013

# compileorc

compileorc — compiles a new orchestra from an ASCII file

## Description

Compileorc will compile one or more instruments at init time, which will be added to the running engine. In case of existing instrument numbers or names, these will be replaced, but any instance still running of the old instrument definition will still perform until it terminates.

## Syntax

```
ires compileorc Sfilename
```

## Initialization

“*Sfilename*” -- a string containing the name of the file containing the orchestra.

“*ires*” -- returns 0 if compilation was successful, or -1 if not.

## Examples

Here is an example of the compileorc opcode. It uses the file *compileorc.csd* [examples/compileorc.csd].

### Example 140. Example of the compileorc opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-o dac -d
</CsOptions>
<CsInstruments>
sr = 44100
nchnls = 1
ksmps = 32
0dbfs = 1

instr 1
ires compileorc "does_not_exist.orc"
print ires ; -1 as could not compile
ires compileorc "my.orc"
print ires ; 0 as compiled successfully
event_i "i", 2, 0, 3, .2, 465 ;send event
endin

</CsInstruments>
<CsScore>
i1 0 1
</CsScore>
</CsoundSynthesizer>
```

## See also

*event*, *event\_i*, *schedule*, *schedwhen*, *schedkwhen*, *schedkwhennamed*

## Credits

Author: Victor Lazzarini, 2013

# compilestr

compilestr — compiles a new orchestra passed in as an ASCII string

## Description

Compilestr will compile one or more instruments at init time, which will be added to the running engine. In case of existing instrument numbers or names, these will be replaced, but any instance still running of the old instrument definition will still perform until it terminates. Only new instances will use the new definition. Multi-line strings are accepted, using `{{ }}` to enclose the string.

## Syntax

```
ires compilestr Sorch
```

## Initialization

“Sorch” -- a string (in double-quotes or enclosed by `{{ }}`) containing one or more instruments.

“ires” -- returns 0 if compilation was successful, or -1 if not.

## Examples

Here is an example of the compilestr opcode. It uses the file `compilestr.csd` [examples/compilestr.csd].

### Example 141. Example of the compilestr opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-o dac -d
</CsOptions>
<CsInstruments>
sr = 44100
nchnls = 1
kmps = 32
0dbfs = 1

instr 1

    ;will fail because of wrong code
ires compilestr {{
instr 2
al oscilb p4, p5, 0
out al
endin
}}
print ires ; returns -1 because not successfull

    ;will compile ...
ires compilestr {{
instr 2
al oscils p4, p5, 0
```

```
out a1
endin
}}
print ires ; ... and returns 0

;call the new instrument
;(note that the overall performance is extended)
scoreline_i "i 2 0 3 .2 415"

endin

</CsInstruments>
<CsScore>
i1 0 1
</CsScore>
</CsoundSynthesizer>
```

## See also

*event, event\_i, schedule, schedwhen, schedkwhen, schedkwhennamed*

## Credits

Author: Victor Lazzarini, 2013

# compress

compress — Compress, limit, expand, duck or gate an audio signal.

## Description

This unit functions as an audio compressor, limiter, expander, or noise gate, using either soft-knee or hard-knee mapping, and with dynamically variable performance characteristics. It takes two audio input signals, *aasig* and *acsig*, the first of which is modified by a running analysis of the second. Both signals can be the same, or the first can be modified by a different controlling signal.

**compress** first examines the controlling *acsig* by performing envelope detection. This is directed by two control values *katt* and *krel*, defining the attack and release time constants (in seconds) of the detector. The detector rides the peaks (not the RMS) of the control signal. Typical values are .01 and .1, the latter usually being similar to *ilook*.

The running envelope is next converted to decibels, then passed through a mapping function to determine what compressor action (if any) should be taken. The mapping function is defined by four decibel control values. These are given as positive values, where 0 db corresponds to an amplitude of 0dbfs/32768, and 90 db corresponds to an amplitude of 0dbfs.

## Syntax

ar **compress** aasig, acsig, kthresh, kloknee, khiknee, kratio, katt, krel, ilook

## Initialization

*ilook* -- lookahead time in seconds, by which an internal envelope release can sense what is coming. This induces a delay between input and output, but a small amount of lookahead improves the performance of the envelope detector. Typical value is .05 seconds, sufficient to sense the peaks of the lowest frequency in *acsig*.

## Performance

*kthresh* -- sets the lowest decibel level that will be allowed through. Normally 0 or less, but if higher the threshold will begin removing low-level signal energy such as background noise.

*kloknee*, *khiknee* -- decibel break-points denoting where compression or expansion will begin. These set the boundaries of a soft-knee curve joining the low-amplitude 1:1 line and the higher-amplitude compression ratio line. Typical values are 48 and 60 db. If the two breakpoints are equal, a hard-knee (angled) map will result.

*kratio* -- ratio of compression when the signal level is above the knee. The value 2 will advance the output just one decibel for every input gain of two; 3 will advance just one in three; 20 just one in twenty, etc. Inverse ratios will cause signal expansion: .5 gives two for one, .25 four for one, etc. The value 1 will result in no change.

The actions of **compress** will depend on the parameter settings given. A hard-knee compressor-limiter, for instance, is obtained from a near-zero attack time, equal-value break-points, and a very high ratio (say 100). A noise-gate plus expander is obtained from some positive threshold, and a fractional ratio above the knee. A voice-activated music compressor (ducker) will result from feeding the music into *aasig* and the speech into *acsig*. A voice de-esser will result from feeding the voice into both, with the *acsig* version

being preceded by a band-pass filter that emphasizes the sibilants. Each application will require some experimentation to find the best parameter settings; these have been made k-variable to make this practical.

## Examples

Here is an example of the compress opcode. It uses the file *compress.csd* [examples/compress.csd].

### Example 142. Example of the compress opcode.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -iadc    ;;RT audio out and in
; For Non-realtime ouput leave only the line below:
; -o compress.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

instr 1 ; uncompressed signal
asig disk2 "beats.wav", 1, 0, 1
outs asig, asig
endin

instr 2 ; compressed signal.
; Use the "beats.wav" audio file and a mic
avoice in
asig disk2 "beats.wav", 1, 0, 1

; duck the audio signal "beats.wav" with your voice.
kthresh = 0
kloknee = 40
khiknee = 60
kratio = 3
katt = 0.1
krel = .5
ilook = .02
asig compress asig, avoice, kthresh, kloknee, khiknee, kratio, katt, krel, ilook ; voice-activated com
outs asig, asig

endin

</CsInstruments>
<CsScore>

i 1 0 5

i 2 6 21

e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*dam*

*compress2*

## **Credits**

Written by Barry L. Vercoe for Extended Csound and released in csound5.02.

# compress2

compress2 — Compress, limit, expand, duck or gate an audio signal.

## Description

This unit functions as an audio compressor, limiter, expander, or noise gate, using either soft-knee or hard-knee mapping, and with dynamically variable performance characteristics. It takes two audio input signals, *aasig* and *acsig*, the first of which is modified by a running analysis of the second. Both signals can be the same, or the first can be modified by a different controlling signal.

**compress2** first examines the controlling *acsig* by performing envelope detection. This is directed by two control values *katt* and *krel*, defining the attack and release time constants (in seconds) of the detector. The detector rides the peaks (not the RMS) of the control signal. Typical values are .01 and .1, the latter usually being similar to *ilook*.

The running envelope is next converted to decibels, then passed through a mapping function to determine what compressor action (if any) should be taken. The mapping function is defined by four decibel control values. These are given as positive values, where 0 db corresponds to an amplitude of 0dbfs.

## Syntax

```
ar compress2 aasig, acsig, kthresh, kloknee, khiknee, kratio, katt, krel, ilook
```

## Initialization

*ilook* -- lookahead time in seconds, by which an internal envelope release can sense what is coming. This induces a delay between input and output, but a small amount of lookahead improves the performance of the envelope detector. Typical value is .05 seconds, sufficient to sense the peaks of the lowest frequency in *acsig*.

## Performance

*kthresh* -- sets the lowest decibel level that will be allowed through. Normally -90 or less, but if higher the threshold will begin removing low-level signal energy such as background noise.

*kloknee*, *khiknee* -- decibel break-points denoting where compression or expansion will begin. These set the boundaries of a soft-knee curve joining the low-amplitude 1:1 line and the higher-amplitude compression ratio line. Typical values are -52 and -30 db. If the two breakpoints are equal, a hard-knee (angled) map will result.

*kratio* -- ratio of compression when the signal level is above the knee. The value 2 will advance the output just one decibel for every input gain of two; 3 will advance just one in three; 20 just one in twenty, etc. Inverse ratios will cause signal expansion: .5 gives two for one, .25 four for one, etc. The value 1 will result in no change.

The actions of **compress2** will depend on the parameter settings given. A hard-knee compressor-limiter, for instance, is obtained from a near-zero attack time, equal-value break-points, and a very high ratio (say 100). A noise-gate plus expander is obtained from some positive threshold, and a fractional ratio above the knee. A voice-activated music compressor (ducker) will result from feeding the music into *aasig* and the speech into *acsig*. A voice de-esser will result from feeding the voice into both, with the *acsig* version

being preceded by a band-pass filter that emphasizes the sibilants. Each application will require some experimentation to find the best parameter settings; these have been made k-variable to make this practical.

## Examples

Here is an example of the `compress2` opcode. It uses the file `compress2.csd` [examples/compress2.csd].

### Example 143. Example of the `compress2` opcode.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -iadc    ;;RT audio out and in
; For Non-realtime ouput leave only the line below:
; -o compress2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

instr 1 ; uncompressed signal
asig disk2 "beats.wav", 1, 0, 1
outs asig, asig
endin

instr 2 ; compressed signal.
; Use the "beats.wav" audio file and a mic
avoice in
asig disk2 "beats.wav", 1, 0, 1

; duck the audio signal "beats.wav" with your voice.
kthresh = -90
kloknee = -50
khiknee = -30
kratio = 3
katt = 0.1
krel = .5
ilook = .02
asig compress2 asig, avoice, kthresh, kloknee, khiknee, kratio, katt, krel, ilook ; voice-activated c
outs asig, asig

endin

</CsInstruments>
<CsScore>

i 1 0 5

i 2 6 21

e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*dam*

*compress*

## Credits

Written by John ffitch after Barry L. Vercoe from Extended but with more common dB values, new in version 6.07.

# connect

connect — Connects a source outlet to a sink inlet.

## Description

The connect opcode, valid only in the orchestra header, sends the signals from the indicated outlet in all instances of the indicated source instrument to the indicated inlet in all instances of the indicated sink instrument. Each inlet instance receives the sum of the signals in all outlet instances. Thus multiple instances of an outlet may fan in to one instance of an inlet, or one instance of an outlet may fan out to multiple instances of an inlet.

When Csound creates a new instance of an instrument template, new instances of its connections also are created.

## Syntax

```
connect Tsource1, Soutlet1, Tsink1, Sinlet1
```

## Initialization

*Tsource1* -- String name of the source instrument definition.

*Soutlet1* -- String name of the source outlet in the source instrument.

*Tsink1* -- String name of the sink instrument definition.

*Sinlet1* -- String name of the sink inlet in the sink instrument.

## Examples

Here is an example of the connect opcode. It uses the file *connect.csd* [examples/connect.csd].

### Example 144. Example of the connect opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o connect.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

/* Written by Michael Gogins */
; Initialize the global variables.
sr = 44100
ksmps = 32
nchnls = 2

; Connect up the instruments to create a signal flow graph.

connect "SimpleSine", "leftout", "Reverberator", "leftin"
connect "SimpleSine", "rightout", "Reverberator", "rightin"
connect "Moogy", "leftout", "Reverberator", "leftin"
```

```

connect "Moogy",      "rightout",  "Reverberator",  "rightin"

connect "Reverberator", "leftout",  "Compressor",   "leftin"
connect "Reverberator", "rightout", "Compressor",   "rightin"

connect "Compressor", "leftout",  "Soundfile",    "leftin"
connect "Compressor", "rightout", "Soundfile",    "rightin"

; Turn on the "effect" units in the signal flow graph.

alwayson "Reverberator", 0.91, 12000
alwayson "Compressor"
alwayson "Soundfile"

instr SimpleSine
  ihz = cpsmidinn(p4)
  iamplitude = ampdb(p5)
  print ihz, iamplitude
  ; Use ftgenonce instead of ftgen, ftgentmp, or f statement.
  isine ftgenonce 0, 0, 4096, 10, 1
  a1 oscili iamplitude, ihz, isine
  aenv madsr 0.05, 0.1, 0.5, 0.2
  asignal = a1 * aenv
  ; Stereo audio outlet to be routed in the orchestra header.
  outleta "leftout", asignal * 0.25
  outleta "rightout", asignal * 0.75
endin

instr Moogy
  ihz = cpsmidinn(p4)
  iamplitude = ampdb(p5)
  ; Use ftgenonce instead of ftgen, ftgentmp, or f statement.
  isine ftgenonce 0, 0, 4096, 10, 1
  asignal vco iamplitude, ihz, 1, 0.5, isine
  kfco line 200, p3, 2000
  krez init 0.9
  asignal moogvcf asignal, kfco, krez, 100000
  ; Stereo audio outlet to be routed in the orchestra header.
  outleta "leftout", asignal * 0.75
  outleta "rightout", asignal * 0.25
endin

instr Reverberator
  ; Stereo input.
  aleftin inleta "leftin"
  arightin inleta "rightin"
  idelay = p4
  icutoff = p5
  aleftout, arightout reverbsc aleftin, arightin, idelay, icutoff
  ; Stereo output.
  outleta "leftout", aleftout
  outleta "rightout", arightout
endin

instr Compressor
  ; Stereo input.
  aleftin inleta "leftin"
  arightin inleta "rightin"
  kthreshold = 25000
  icomp1 = 0.5
  icomp2 = 0.763
  irtime = 0.1
  iftime = 0.1
  aleftout dam aleftin, kthreshold, icomp1, icomp2, irtime, iftime
  arightout dam arightin, kthreshold, icomp1, icomp2, irtime, iftime
  ; Stereo output.
  outleta "leftout", aleftout

```

```
outleta "rightout", arightout
endin

instr Soundfile
; Stereo input.
aleftin inleta "leftin"
arightin inleta "rightin"
outs aleftin, arightin
endin

</CsInstruments>
<CsScore>
; Not necessary to activate "effects" or create f-tables in the score!
; Overlapping notes to create new instances of instruments.
i "SimpleSine" 1 5 60 85
i "SimpleSine" 2 5 64 80
i "Moogy" 3 5 67 75
i "Moogy" 4 5 71 70
;6 extra seconds after the performance
e 12
</CsScore>
</CsoundSynthesizer>
```

## See Also

*outleta outletk outletf inleta inletk inletf alwayson ftgenonce*

More information on this opcode: <http://www.csoundjournal.com/issue13/signalFlowGraphOpcodes.html>  
, written by Michael Gogins

## Credits

By: Michael Gogins 2009

# control

`control` — Configurable slider controls for realtime user input.

## Description

Configurable slider controls for realtime user input. Requires Winsound or TCL/TK. *control* reads a slider's value.

## Syntax

```
kres control knum
```

## Performance

Note that this opcode is not available on Windows due to the implimentation of pipes on that system

*knnum* -- number of the slider to be read.

Calling *control* will create a new slider on the screen. There is no theoretical limit to the number of sliders. Windows and TCL/TK use only integers for slider values, so the values may need rescaling. GUIs usually pass values at a fairly slow rate, so it may be advisable to pass the output of *control* through *port*.

## Examples

See the *setctrl* opcode for an example.

## See Also

*setctrl*

## Credits

Author: John ffitch  
University of Bath, Codemist. Ltd.  
Bath, UK  
May, 2000

New in Csound version 4.06

## convle

convle — Same as the convolve opcode.

### Description

Same as the *convolve* opcode.

# convolve

convolve — Convolves a signal and an impulse response.

## Description

Output is the convolution of signal *ain* and the impulse response contained in *ifilcod*. If more than one output signal is supplied, each will be convolved with the same impulse response. Note that it is considerably more efficient to use one instance of the operator when processing a mono input to create stereo, or quad, outputs.

Note: this opcode can also be written as *convle*.

## Syntax

```
ar1 [, ar2] [, ar3] [, ar4] convolve ain, ifilcod [, ichannel]
```

## Initialization

*ifilcod* -- integer or character-string denoting an impulse response data file. An integer denotes the suffix of a file *convolve.m*; a character string (in double quotes) gives a filename, optionally a full pathname. If not a fullpath, the file is sought first in the current directory, then in the one given by the environment variable SADIR (if defined). The data file contains the Fourier transform of an impulse response. Memory usage depends on the size of the data file, which is read and held entirely in memory during computation, but which is shared by multiple calls.

*ichannel* (optional) -- which channel to use from the impulse response data file.

## Performance

*ain* -- input audio signal.

*convolve* implements Fast Convolution. The output of this operator is delayed with respect to the input. The following formulas should be used to calculate the delay:

```
For (1/kr) <= IRdur:
    Delay = ceil(IRdur * kr) / kr
For (1/kr) > IRdur:
    Delay = IRdur * ceil(1/(kr*IRdur))
Where:
    kr = Csound control rate
    IRdur = duration, in seconds, of impulse response
    ceil(n) = smallest integer not smaller than n
```

One should be careful to also take into account the initial delay, if any, of the impulse response. For example, if an impulse response is created from a recording, the soundfile may not have the initial delay included. Thus, one should either ensure that the soundfile has the correct amount of zero padding at the start, or, preferably, compensate for this delay in the orchestra (the latter method is more efficient). To compensate for the delay in the orchestra, subtract the initial delay from the result calculated using the above formula(s), when calculating the required delay to introduce into the 'dry' audio path.

For typical applications, such as reverb, the delay will be in the order of 0.5 to 1.5 seconds, or even longer. This renders the current implementation unsuitable for real time applications. It could conceivably be used for real time filtering however, if the number of taps is small enough.

The author intends to create a higher-level operator at some stage, that would mix the wet & dry signals, using the correct amount of delay automatically.

## Examples

Create frequency domain impulse response file using the *cvanal utility*:

```
csound -Ucvanal 11_44.wav 11_44.cv
```

Determine duration of impulse response. For high accuracy, determine the number of sample frames in the impulse response soundfile, and then compute the duration with:

$$\text{duration} = (\text{sample frames}) / (\text{sample rate of soundfile})$$

This is due to the fact that the *sndinfo utility* only reports the duration to the nearest 10ms. If you have a utility that reports the duration to the required accuracy, then you can simply use the reported value directly.

```
sndinfo 11_44.wav
```

length = 60822 samples, sample rate = 44100

Duration = 60822/44100 = 1.379s.

Determine initial delay, if any, of impulse response. If the impulse response has not had the initial delay removed, then you can skip this step. If it has been removed, then the only way you will know the initial delay is if the information has been provided separately. For this example, let's assume that the initial delay is 60ms (0.06s)

Determine the required delay to apply to the dry signal, to align it with the convolved signal:

If  $kr = 441$ :

$1/kr = 0.0023$ , which is  $\leq IRdur$  (1.379s), so:

$$\begin{aligned} \text{Delay1} &= \text{ceil}(IRdur * kr) / kr \\ &= \text{ceil}(608.14) / 441 \\ &= 609/441 \\ &= 1.38\text{s} \end{aligned}$$

Accounting for the initial delay:

$$\begin{aligned} \text{Delay2} &= 0.06\text{s} \\ \text{Total delay} &= \text{delay1} - \text{delay2} \\ &= 1.38 - 0.06 \\ &= 1.32\text{s} \end{aligned}$$

Here is similar example of the convolve opcode. It uses the file *convolve.csd* [examples/convolve.csd].

### Example 145. Example of the convolve opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>
```

```

; Select audio/midi flags here according to platform
;-odac      ;;RT audio out
-iadc      ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
;-o convolve.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
; NB: 'Small' reverbs often require a much higher
; percentage of wet signal to sound interesting. 'Large'
; reverbs seem require less. Experiment! The wet/dry mix is
; very important - a small change can make a large difference.

sr = 44100
ksmps = 32
nchnls = 2
Odbfs = 1
;The analysis file is not system independent!
; create "rv_mono.wav" and "rv_stereo.wav" with cvanal first!

instr 1

imix = 0.25 ;wet/dry mix. Vary as desired.
ivol = 1 ;Overall volume level of reverb. May need to adjust
;when wet/dry mix is changed, to avoid clipping.

idel  filelen p4 ;calculate length and number of channels of soundfile
print idel
ichnls filenchnls p4
print ichnls

if (ichnls == 1) then

adry  soundin "fox.wav" ; input (dry) audio
awet  convolve adry,"rv_mono.cva" ; mono convolved (wet) audio
awet  diff awet ; brighten
adrydel delay (1-imix)*adry, idel ; Delay dry signal to align it with convolved signal
; Apply level adjustment here too.
outs  ivol*(adrydel+imix*awet),ivol*(adrydel+imix*awet) ; Mix wet & dry

else

adry  soundin "fox.wav" ; input (dry) audio
awet1, awet2 convolve adry,"rv_stereo.cva" ; stereo convolved (wet) audio
awet1  diff awet1 ; brighten left
awet2  diff awet2 ; and brighten right
adrydel delay (1-imix)*adry, idel ; Delay dry signal to align it with convolved signal
; Apply level adjustment here too.
outs  ivol*(adrydel+imix*awet1),ivol*(adrydel+imix*awet2) ; Mix wet & dry signals

endif

endin

</CsInstruments>
<CsScore>

i 1 0 4 "rv_mono.wav"
i 1 5 4 "rv_stereo.wav"

e
</CsScore>
</CsoundSynthesizer>

```

## See also

*pconvolve*, *dconv*, *cvanal*.

## Credits

Author: Greg Sullivan

1996

New in version 3.28

# copya2ftab

copya2ftab — Copy data from a vector to an f-table.

## Description

The *copya2ftab* opcode takes a k-array and copies the contents to an f-table.

## Syntax

```
copya2ftab kftbl, tab
```

## Performance

*kftbl* -- one-dimensional array for source.

*tab* -- f-table for destination.

## Example

Here is an example of the *copya2ftab* opcode. It uses the file *copya2ftab.csd* [examples/copya2ftab.csd].

### Example 146. Example of the *copya2ftab* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

;an 'empty' function table with 10 points
giTable ftgen 0, 0, -10, 2, 0

instr 1

;print initial values of giTable
puts      "\nInitial table content:", 1
indx     = 0
until indx == ftlen(giTable) do
iVal     table  indx, giTable
printf_i "Table index %d = %f\n", 1, indx, iVal
indx += 1
od

;create array
kArr[]   init 10
```

```
;fill in values
kArr   genarray 1, 10

;print array values
printf "%s", 1, "\nArray content:\n"
kndx   = 0
  until kndx == lenarray(kArr) do
    printf "kArr[%d] = %f\n", kndx+1, kndx, kArr[kndx]
kndx   += 1
  od

;copy array values to table
copya2ftab kArr, giTable

;print modified values of giTable
printf "%s", 1, "\nModified table content after copya2ftab:\n"
kndx   = 0
  until kndx == ftlen(giTable) do
    kVal table kndx, giTable
    printf "Table index %d = %f\n", kndx+1, kndx, kVal
kndx   += 1
  od

;turn instrument off
turnoff
endin

</CsInstruments>
<CsScore>
i 1 0 0.1
</CsScore>
</CsoundSynthesizer>
```

## See Also

*copyf2array*

## Credits

Author: John ffitch  
October 2011

New in Csound version 5.15

Renamed in Csound version 6.00

# copyf2array

copyf2array — Copy data from an f-table to a vector.

## Description

The *copyf2array* opcode takes an f-table and copies the contents to a t-var.

## Syntax

```
copyf2array tab, kftbl
```

## Performance

*tab* -- tables for destination.

*kftbl* -- f-tables for source.

## Example

Here is an example of the copyf2array opcode. It uses the file *copyf2array.csd* [examples/copyf2array.csd].

### Example 147. Example of the copyf2array opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

;8 points sine wave function table
giSine ftgen 0, 0, 8, 10, 1

instr 1
;create array
kArr[] init 8

;copy table values in it
copyf2array kArr, giSine

;print values
kndx = 0
until kndx == lenarray(kArr) do
printf "kArr[%d] = %f\n", kndx+1, kndx, kArr[kndx]
kndx += 1
od
```

```
;turn instrument off
  turnoff
endin

</CsInstruments>
<CsScore>
i 1 0 0.1
</CsScore>
</CsoundSynthesizer>
```

## See Also

*copya2ftab*

## Credits

Author: John ffitch  
October 2011

New in Csound version 5.15

Renamed in Csound version 6.00

# COS

cos — Performs a cosine function.

## Description

Returns the cosine of  $x$  ( $x$  in radians).

## Syntax

```
cos(x) (no rate
      restriction)

cos(k/i[]) (k- or i-arrays )
```

## Examples

Here is an example of the cos opcode. It uses the file *cos.csd* [examples/cos.csd].

### Example 148. Example of the cos opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o cos.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
icos1 =      cos(0) ;cosine of 0 is 1
icos2 =      cos($M_PI_2) ;cosine of pi/2 (1.5707...) is 0
icos3 =      cos($M_PI) ;cosine of pi (3.1415...) is -1
icos4 =      cos($M_PI_2 * 3) ;cosine of 3/2pi (4.7123...) is 0
icos5 =      cos($M_PI * 2) ;cosine of 2pi (6.2831...) is 1
icos6 =      cos($M_PI * 4) ;cosine of 4pi is also 1 as it is periodically to 2pi
print      icos1, icos2, icos3, icos4, icos5, icos6
endin

instr 2 ;cos used in panning, after an example from Hans Mikelson
aout      vco2      0.8, 220 ; sawtooth
kpan      linseg    p4, p3, p5 ;0 = left, 1 = right
kpan      =         kpan*$M_PI_2 ;range 0-1 becomes 0-pi/2
kpanl     =         cos(kpan)
kpanr     =         sin(kpan)
outs      =         aout*kpanl, aout*kpanr
endin
```

```
</CsInstruments>
<CsScore>
i 1 0 0
i 2 0 5 0 1 ;move left to right
i 2 6 5 1 0 ;move right to left
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*cosh, cosinv, sin, sinh, sininv, tan, tanh, taninv*

# cosseg

cosseg — Trace a series of line segments between specified points with cosine interpolation.

## Description

Trace a series of line segments between specified points with cosine interpolation.

## Syntax

```
ares cosseg ia, idur1, ib [, idur2] [, ic] [...]  
kres cosseg ia, idur1, ib [, idur2] [, ic] [...]
```

## Initialization

*ia* -- starting value.

*ib*, *ic*, etc. -- value after *dur1* seconds, etc.

*idur1* -- duration in seconds of first segment. A zero or negative value will cause all initialization to be skipped.

*idur2*, *idur3*, etc. -- duration in seconds of subsequent segments. A zero or negative value will terminate the initialization process with the preceding point, permitting the last-defined line or curve to be continued indefinitely in performance. The default is zero.

## Performance

These units generate control or audio signals whose values can pass through 2 or more specified points. The sum of *dur* values may or may not equal the instrument's performance time: a shorter performance will truncate the specified pattern, while a longer one will cause the last value to be repeated until the end of the note.

## Examples

Here is an example of the cosseg opcode. It uses the file *cosseg.csd* [examples/cosseg.csd].

### Example 149. Example of the cosseg opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
-odac      ;;realtime audio out  
;-iadc     ;;uncomment -iadc if realtime audio input is needed too  
; For Non-realtime ouput leave only the line below:  
;-o linseg.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>
```

```

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine ftgen 0, 0, 2^10, 10, 1

instr 1

kcps = cpspch(p4)
kenv cosseg 0, 0.25, 1, 0.75, 0 ; together = 1 sec
asig poscil kenv, kcps, giSine
outs asig, asig

endin

instr 2 ; scaling to duration

kcps = cpspch(p4)
kenv cosseg 0, p3*0.25, 1, p3*0.75, 0
asig poscil kenv, kcps, giSine
outs asig, asig

endin

instr 3 ; with negative value

kcps = cpspch(p4)
aenv cosseg 0, 0.1, 1, 0.5, -0.9, 0.4, 0
asig poscil aenv, kcps, giSine
outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 1 7.00 ; = 1 sec, p3 fits exactly
i 1 2 2 7.00 ; = 2 sec, p3 truncated at 1 sec

i 2 4 1 7.00 ; scales to duration
i 2 6 2 7.00 ; of p3

i 3 9 2 7.00
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*expon, expseg, expsegb, expsegr, line, linseg, linsegr transeg transegb*

## Credits

Author: John ffitth  
Aug 2012

New in version 5.18

# cossegb

cossegb — Trace a series of line segments between specified absolute points with cosine interpolation.

## Description

Trace a series of line segments between specified absolute points with cosine interpolation.

## Syntax

```
ares cossegb ia, itim1, ib [, itim2] [, ic] [...]  
kres cossegb ia, itim1, ib [, itim2] [, ic] [...]
```

## Initialization

*ia* -- starting value.

*ib*, *ic*, etc. -- value at *tim1* seconds, etc.

*itim1* -- time in seconds of end of first segment. A zero or negative value will cause all initialization to be skipped.

*itim2*, *itim3*, etc. -- time in seconds at the end of subsequent segments.

## Performance

These units generate control or audio signals whose values can pass through 2 or more specified points. The last *tim* value may or may not equal the instrument's performance time: a shorter performance will truncate the specified pattern, while a longer one will cause the last value to be repeated until the end of the note.

## Examples

Here is an example of the cossegb opcode. It uses the file *cossegb.csd* [examples/cossegb.csd].

### Example 150. Example of the cossegb opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
-odac      ;;realtime audio out  
;-iadc     ;;uncomment -iadc if realtime audio input is needed too  
; For Non-realtime ouput leave only the line below:  
;-o linseg.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>  
  
sr = 44100  
ksmps = 32  
nchnls = 2
```

```
Odbfs = 1

giSine ftgen 0, 0, 2^10, 10, 1

instr 1

kcps = cpspch(p4)
kenv cossegb 0, 0.25, 1, 1, 0
asig poscil kenv, kcps, giSine
      outs asig, asig

endin

instr 2 ; scaling to duration

kcps = cpspch(p4)
kenv cossegb 0, p3*0.25, 1, p3, 0
asig poscil kenv, kcps, giSine
      outs asig, asig

endin

</CsInstruments>
<CsScore>

i 1 0 1 7.00 ; = 1 sec, p3 fits exactly
i 1 2 2 7.00 ; = 2 sec, p3 truncated at 1 sec

i 2 4 1 7.00 ; scales to duration
i 2 6 2 7.00 ; of p3

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*cosseg, expon, expseg, expsegr, line, linseg linsegb linsegr transeg*

## Credits

Author: John ffitch  
Aug 2012

New in version 5.18

## cossegr

`cossegr` — Trace a series of line segments between specified points with cosine interpolation, including a release segment.

### Description

Trace a series of line segments between specified points with cosine interpolation, including a release segment.

### Syntax

```
ares cossegr ia, idur1, ib [, idur2] [, ic] [...], irel, iz  
kres cossegr ia, idur1, ib [, idur2] [, ic] [...], irel, iz
```

### Initialization

*ia* -- starting value.

*ib*, *ic*, etc. -- value after *dur1* seconds, etc.

*idur1* -- duration in seconds of first segment. A zero or negative value will cause all initialization to be skipped.

*idur2*, *idur3*, etc. -- duration in seconds of subsequent segments. A zero or negative value will terminate the initialization process with the preceding point, permitting the last-defined line or curve to be continued indefinitely in performance. The default is zero.

*irel*, *iz* -- duration in seconds and final value of a note releasing segment.

### Performance

These units generate control or audio signals whose values can pass through 2 or more specified points. The sum of *dur* values may or may not equal the instrument's performance time: a shorter performance will truncate the specified pattern, while a longer one will cause the last-defined segment to continue on in the same direction.

`cossegr` is amongst the Csound “r” units that contain a note-off sensor and release time extender. When each senses an event termination or MIDI noteoff, it immediately extends the performance time of the current instrument by *irel* seconds, and sets out to reach the value *iz* by the end of that period (no matter which segment the unit is in). “r” units can also be modified by MIDI noteoff velocities. For two or more extenders in an instrument, extension is by the greatest period.

You can use other pre-made envelopes which start a release segment upon receiving a note off message, like `linenr` and `expsegr`, or you can construct more complex envelopes using `xtratim` and `release`. Note that you don't need to use `xtratim` if you are using `cossegr`, since the time is extended automatically.

### Examples

Here is an example of the `cossegr` opcode. It uses the file `cossegr.csd` [examples/cossegr.csd].

### Example 151. Example of the cossegr opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0   ;;realtime audio out and realtime midi in
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o linsegr.wav -W   ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

icps cpsmidi
iamp ampmidi .3

kenv cossegr 1, .05, 0.5, 1, 0
asig pluck kenv, icps, 200, 1, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 4096 10 1 ;sine wave

f0 30 ;runs 30 seconds
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*linenr, linsegr, expsegr, envlpxr, mxadsr, madsr expon, expseg, expsega line, linseg, xtratim, transegr*

## Credits

Author: John ffitth

August 2012.

New in Csound 5.18

# cosh

cosh — Performs a hyperbolic cosine function.

## Description

Returns the hyperbolic cosine of  $x$  ( $x$  in radians).

## Syntax

```
cosh(x) (no rate  
restriction)  
  
cosh(k/i[]) (k- or i-arrays )
```

## Examples

Here is an example of the cosh opcode. It uses the file *cosh.csd* [examples/cosh.csd].

### Example 152. Example of the cosh opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
; Audio out  Audio in  
-odac          -iadc      ;;RT audio I/O  
; For Non-realtime ouput leave only the line below:  
; -o cosh.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>  
  
; Initialize the global variables.  
sr = 44100  
kr = 4410  
ksmps = 10  
nchnls = 1  
  
; Instrument #1.  
instr 1  
  irad = 1  
  i1 = cosh(irad)  
  
  print i1  
endin  
  
</CsInstruments>  
<CsScore>  
  
; Play Instrument #1 for one second.  
i 1 0 1  
e  
  
</CsScore>
```

```
</CsoundSynthesizer>
```

Its output should include lines like this:

```
instr 1: i1 = 1.543
```

## See Also

*cos, cosinv, sin, sinh, sininv, tan, tanh, taninv*

## Credits

Author: John ffitch

New in version 3.47

Example written by Kevin Conder.

# cosinv

cosinv — Performs an arccosine function.

## Description

Returns the arccosine of  $x$  ( $x$  in radians).

## Syntax

```
cosinv(x) (no rate  
restriction)  
cosinv(k/i[]) (k- or i-arrays )
```

## Examples

Here is an example of the cosinv opcode. It uses the file *cosinv.csd* [examples/cosinv.csd].

### Example 153. Example of the cosinv opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
; Audio out  Audio in  
-odac          -iadc      ;;RT audio I/O  
; For Non-realtime output leave only the line below:  
; -o cosinv.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>  
  
; Initialize the global variables.  
sr = 44100  
kr = 4410  
ksmps = 10  
nchnls = 1  
  
; Instrument #1.  
instr 1  
  irad = 0.5  
  i1 = cosinv(irad)  
  
  print i1  
endin  
  
</CsInstruments>  
<CsScore>  
  
; Play Instrument #1 for one second.  
i 1 0 1  
e  
  
</CsScore>
```

```
</CsoundSynthesizer>
```

Its output should include lines like this:

```
instr 1: i1 = 1.047
```

## See Also

*cos, cosh, sin, sinh, sininv, tan, tanh, taninv*

## Credits

Author: John ffitch

New in version 3.48

Example written by Kevin Conder.

# cps2pch

`cps2pch` — Converts a pitch-class value into cycles-per-second (Hz) for equal divisions of the octave.

## Description

Converts a pitch-class value into cycles-per-second (Hz) for equal divisions of the octave.

## Syntax

```
icps cps2pch ipch, iequal
```

## Initialization

`ipch` -- Input number of the form `8ve.pc`, indicating an 'octave' and which note in the octave.

`iequal` -- If positive, the number of equal intervals into which the 'octave' is divided. Must be less than or equal to 100. If negative, is the number of a table of frequency multipliers.



### Note

1. The following are essentially the same

```
ia = cpspch(8.02)
ib  cps2pch 8.02, 12
ic  cpsxpch 8.02, 12, 2, 1.02197503906
```

2. These are opcodes not functions
3. Negative values of `ipch` are allowed.

## Examples

Here is an example of the `cps2pch` opcode. It uses the file `cps2pch.csd` [examples/cps2pch.csd].

### Example 154. Example of the `cps2pch` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o cps2pch.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
```

```

ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Use a normal twelve-tone scale.
  ipch = 8.02
  iequal = 12

  icps cps2pch ipch, iequal

  print icps
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
e

</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like this:

```
instr 1: icps = 293.666
```

Here is an example of the `cps2pch` opcode using a table of frequency multipliers. It uses the file `cps2pch_ftable.csd` [examples/cps2pch\_ftable.csd].

### Example 155. Example of the `cps2pch` opcode using a table of frequency multipliers.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o cps2pch_ftable.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ipch = 8.02

  ; Use Table #1, a table of frequency multipliers.
  icps cps2pch ipch, -1

  print icps
endin

</CsInstruments>
<CsScore>

```

```

; Table #1: a table of frequency multipliers.
; Creates a 10-note scale of unequal divisions.
f 1 0 16 -2 1 1.1 1.2 1.3 1.4 1.6 1.7 1.8 1.9

; Play Instrument #1 for one second.
i 1 0 1
e

```

```

</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like this:

```
instr 1: icps = 313.951
```

Here is an example of the `cps2pch` opcode using a 19ET scale. It uses the file `cps2pch_19et.csd` [examples/cps2pch\_19et.csd].

### Example 156. Example of the `cps2pch` opcode using a 19ET scale.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in
-odac         -iadc    ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o cps2pch_19et.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Use 19ET scale.
ipch = 8.02
iequal = 19

icps cps2pch ipch, iequal

print icps
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
e

</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like this:

```
instr 1: icps = 281.429
```

## See Also

*cpspch, cpsxpch*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
1997

New in Csound version 3.492

# cpsmidi

`cpsmidi` — Get the note number of the current MIDI event, expressed in cycles-per-second.

## Description

Get the note number of the current MIDI event, expressed in cycles-per-second.

## Syntax

```
icps cpsmidi
```

## Performance

Get the note number of the current MIDI event, expressed in cycles-per-second units, for local processing.



### **cpsmidi vs. cpsmidinn**

The `cpsmidi` opcode only produces meaningful results in a Midi-activated note (either real-time or from a Midi score with the -F flag). With `cpsmidi`, the Midi note number value is taken from the Midi event that is internally associated with the instrument instance. On the other hand, the `cpsmidinn` opcode may be used in any Csound instrument instance whether it is activated from a Midi event, score event, line event, or from another instrument. The input value for `cpsmidinn` might for example come from a p-field in a textual score or it may have been retrieved from the real-time Midi event that activated the current note using the `notnum` opcode.

## Examples

Here is an example of the `cpsmidi` opcode. It uses the file `cpsmidi.csd` [examples/cpsmidi.csd].

### **Example 157. Example of the cpsmidi opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      -M0   ;;RT audio I/O with MIDI in
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o cpsmidi.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
icps cpsmidi
```

```
asig oscil 0.6, icps, 1
print icps
outs asig, asig

endin

</CsInstruments>
<CsScore>
f0 20
;sine wave.
f 1 0 16384 10 1

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*aftouch, ampmidi, cpsmidib, cpstmid, midictrl, notnum, octmidi, octmidib, pchbend, pchmidi, pchmidib, veloc, cpsmidinn, octmidinn, pchmidinn*

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

# cpsmidib

`cpsmidib` — Get the note number of the current MIDI event and modify it by the current pitch-bend value, express it in cycles-per-second.

## Description

Get the note number of the current MIDI event and modify it by the current pitch-bend value, express it in cycles-per-second.

## Syntax

```
icps cpsmidib [irange]
kcps cpsmidib [irange]
```

## Initialization

*irange* (optional) -- the pitch bend range in semitones.

## Performance

Get the note number of the current MIDI event, modify it by the current pitch-bend value, and express the result in cycles-per-second units. Available as an i-time value or as a continuous k-rate value.

## Examples

Here is an example of the `cpsmidib` opcode. It uses the file `cpsmidib.csd` [examples/cpsmidib.csd].

### Example 158. Example of the `cpsmidib` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      -MO    ;;RT audio I/O with MIDI in
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o cpsmidi.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; move pitch bend wheel while you play

kcps cpsmidib
asig oscil 0.6, kcps, 1
printk2 kcps
```

```
outs asig, asig

endin

</CsInstruments>
<CsScore>
f0 20
; sine wave.
f 1 0 16384 10 1

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*aftouch, ampmidi, cpsmidi, midictrl, notnum, octmidi, octmidib, pchbend, pchmidi, pchmidib, veloc*

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

# cpsmidinn

`cpsmidinn` — Converts a Midi note number value to cycles-per-second.

## Description

Converts a Midi note number value to cycles-per-second.

## Syntax

`cpsmidinn` (MidiNoteNumber) (init- or control-rate args only)

where the argument within the parentheses may be a further expression.

## Performance

`cpsmidinn` is a function that takes an i-rate or k-rate value representing a Midi note number and returns the equivalent frequency value in cycles-per-second (Hertz). This conversion assumes that Middle C is Midi note number 60 and that Middle A is tuned to 440 Hz. Midi note number values are typically integers in the range from 0 to 127 but fractional values or values outside of this range will be interpreted consistently.



### `cpsmidinn` vs. `cpsmidi`

The `cpsmidinn` opcode may be used in any Csound instrument instance whether it is activated from a Midi event, score event, line event, or from another instrument. The input value for `cpsmidinn` might for example come from a p-field in a textual score or it may have been retrieved from the real-time Midi event that activated the current note using the `notnum` opcode. You must specify an i-rate or k-rate expression for the Midi note number that is to be converted. On the other hand, the `cpsmidi` opcode only produces meaningful results in a Midi-activated note (either real-time or from a Midi score with the -F flag). With `cpsmidi`, the Midi note number value is taken from the Midi event associated with the instrument instance, and no location or expression for this value may be specified.

`cpsmidinn` and its related opcodes are really *value converters* with a special function of manipulating pitch data.

Data concerning pitch and frequency can exist in any of the following forms:

**Table 6. Pitch and Frequency Values**

Name	Abbreviation
octave point pitch-class (8ve.pc)	pch
octave point decimal	oct
cycles per second	cps
Midi note number (0-127)	midinn

The first two forms consist of a whole number, representing octave registration, followed by a specially interpreted fractional part. For `pch`, the fraction is read as two decimal digits representing the 12 equal-tempered pitch classes from .00 for C to .11 for B. For `oct`, the fraction is interpreted as a true decimal

fractional part of an octave. The two fractional forms are thus related by the factor 100/12. In both forms, the fraction is preceded by a whole number octave index such that 8.00 represents Middle C, 9.00 the C above, etc. Midi note number values range between 0 and 127 (inclusively) with 60 representing Middle C, and are usually whole numbers. Thus A440 can be represented alternatively by 440 (*cps*), 69 (*midinn*), 8.09 (*pch*), or 8.75 (*oct*). Microtonal divisions of the *pch* semitone can be encoded by using more than two decimal places.

The mnemonics of the pitch conversion units are derived from morphemes of the forms involved, the second morpheme describing the source and the first morpheme the object (result). Thus *cpspch*(8.09) will convert the pitch argument 8.09 to its *cps* (or Hertz) equivalent, giving the value of 440. Since the argument is constant over the duration of the note, this conversion will take place at i-time, before any samples for the current note are produced.

By contrast, the conversion *cpsoct*(8.75 + k1) which gives the value of A440 transposed by the octave interval *k1*. The calculation will be repeated every *k*-period since that is the rate at which *k1* varies.



## Note

The conversion from *pch*, *oct*, or *midinn* into *cps* is not a linear operation but involves an exponential process that could be time-consuming when executed repeatedly. Csound now uses a built-in table lookup to do this efficiently, even at audio rates. Because the table index is truncated without interpolation, pitch resolution when using one of these opcodes is limited to 8192 discrete and equal divisions of the octave, and some pitches of the standard 12-tone equally-tempered scale are very slightly mistuned (by at most 0.15 cents).

## Examples

Here is an example of the *cpsmidinn* opcode. It uses the file *cpsmidinn.csd* [examples/cpsmidinn.csd].

### Example 159. Example of the *cpsmidinn* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
; Prints a table showing the equivalents of all Midi
; note numbers from 0-127 in cycles-per-second,
; octave.decimal, and octave.pitchclass units.

<CsOptions>
; Select audio/midi flags here according to platform.
; This example produces no audio, so we render in
; non-realtime and turn off sound to disk:
-n
</CsOptions>
<CsInstruments>

instr 1
; i-time loop to print conversion table
imidiNN = 0
loop1:
icps = cpsmidinn(imidiNN)
ioct = octmidinn(imidiNN)
ipch = pchmidinn(imidiNN)

print imidiNN, icps, ioct, ipch

imidiNN = imidiNN + 1
```

```

    if (imidiNN < 128) igoto loop1
  endin

instr 2
  ; test k-rate converters
  kMiddleC = 60
  kcps = cpsmidinn(kMiddleC)
  koct = octmidinn(kMiddleC)
  kpch = pchmidinn(kMiddleC)

  printks "%d %f %f %f\n", 1.0, kMiddleC, kcps, koct, kpch
endin

</CsInstruments>
<CsScore>
i1 0 0
i2 0 0.1
e

</CsScore>
</CsoundSynthesizer>

```

Here is another example of the cpsmidinn opcode. It uses the file *cpsmidinn2.csd* [examples/cpsmidinn2.csd].

### Example 160. Second example of the cpsmidinn opcode.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
;;;RT audio out, midi in, note=p4 and velocity=p5
-odac -+rtmidi=virtual -M0d --midi-key=4 --midi-velocity-amp=5
;-iadc ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o cpsmidinn.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

massign 0, 1 ;assign all midi to instr. 1

instr 1 ;play virtual keyboard

inote = p4
icps = cpsmidinn(inote)
asig oscil 0.6, icps, 1
  print icps
  outs asig, asig

endin

</CsInstruments>
<CsScore>
f0 20
;sine wave.
f 1 0 16384 10 1
;play note from score too
i1 0 1 60
e

</CsScore>
</CsoundSynthesizer>

```

## See Also

*octmidinn, pchmidinn, cpsmidi, notnum, cpspch, cpsoct, octcps, octpch, pchoct*

## Credits

Derived from original value converters by Barry Vercoe.

New in version 5.07

# cpsoct

`cpsoct` — Converts an octave-point-decimal value to cycles-per-second.

## Description

Converts an octave-point-decimal value to cycles-per-second.

## Syntax

```
cpsoct (oct) (no rate restriction)
```

where the argument within the parentheses may be a further expression.

## Performance

`cpsoct` and its related opcodes are really *value converters* with a special function of manipulating pitch data.

Data concerning pitch and frequency can exist in any of the following forms:

**Table 7. Pitch and Frequency Values**

Name	Abbreviation
octave point pitch-class (8ve.pc)	pch
octave point decimal	oct
cycles per second	cps
Midi note number (0-127)	midinn

The first two forms consist of a whole number, representing octave registration, followed by a specially interpreted fractional part. For *pch*, the fraction is read as two decimal digits representing the 12 equal-tempered pitch classes from .00 for C to .11 for B. For *oct*, the fraction is interpreted as a true decimal fractional part of an octave. The two fractional forms are thus related by the factor 100/12. In both forms, the fraction is preceded by a whole number octave index such that 8.00 represents Middle C, 9.00 the C above, etc. Midi note number values range between 0 and 127 (inclusively) with 60 representing Middle C, and are usually whole numbers. Thus A440 can be represented alternatively by 440 (*cps*), 69 (*midinn*), 8.09 (*pch*), or 8.75 (*oct*). Microtonal divisions of the *pch* semitone can be encoded by using more than two decimal places.

The mnemonics of the pitch conversion units are derived from morphemes of the forms involved, the second morpheme describing the source and the first morpheme the object (result). Thus `cpspch(8.09)` will convert the pitch argument 8.09 to its *cps* (or Hertz) equivalent, giving the value of 440. Since the argument is constant over the duration of the note, this conversion will take place at *i*-time, before any samples for the current note are produced.

By contrast, the conversion `cpsoct(8.75 + k1)` which gives the value of A440 transposed by the octave interval *k1*. The calculation will be repeated every *k*-period since that is the rate at which *k1* varies.



### Note

The conversion from *pch*, *oct*, or *midinn* into *cps* is not a linear operation but involves an exponential process that could be time-consuming when executed repeatedly. Csound now

uses a built-in table lookup to do this efficiently, even at audio rates. Because the table index is truncated without interpolation, pitch resolution when using one of these opcodes is limited to 8192 discrete and equal divisions of the octave, and some pitches of the standard 12-tone equally-tempered scale are very slightly mistuned (by at most 0.15 cents).

## Examples

Here is an example of the `cpsoct` opcode. It uses the file `cpsoct.csd` [examples/cpsoct.csd].

### Example 161. Example of the `cpsoct` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o cpsoct.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; Convert octave-point-decimal value into Hz

ioct = p4
icps = cpsoct(ioct)
print icps
asig oscil 0.7, icps, 1
outs asig, asig
endin

</CsInstruments>
<CsScore>
;sine wave.
f 1 0 16384 10 1

i 1 0 1 8.75
i 1 + 1 8.77
i 1 + 1 8.79
i 1 + .5 6.30

e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
instr 1: icps = 440.000
instr 1: icps = 446.110
instr 1: icps = 452.344
instr 1: icps = 80.521
```

## See Also

*cspch, octcps, octpch, pchoct, cpsmidinn, octmidinn, pchmidinn*

# cpspch

cpspch — Converts a pitch-class value to cycles-per-second.

## Description

Converts a pitch-class value to cycles-per-second.

## Syntax

`cpspch` (*pch*) (*init-* or *control-rate* args only)

where the argument within the parentheses may be a further expression.

## Performance

*cpspch* and its related opcodes are really *value converters* with a special function of manipulating pitch data.

Data concerning pitch and frequency can exist in any of the following forms:

**Table 8. Pitch and Frequency Values**

Name	Abbreviation
octave point pitch-class (8ve.pc)	pch
octave point decimal	oct
cycles per second	cps
Midi note number (0-127)	midinn

The first two forms consist of a whole number, representing octave registration, followed by a specially interpreted fractional part. For *pch*, the fraction is read as two decimal digits representing the 12 equal-tempered pitch classes from .00 for C to .11 for B. For *oct*, the fraction is interpreted as a true decimal fractional part of an octave. The two fractional forms are thus related by the factor 100/12. In both forms, the fraction is preceded by a whole number octave index such that 8.00 represents Middle C, 9.00 the C above, etc. Midi note number values range between 0 and 127 (inclusively) with 60 representing Middle C, and are usually whole numbers. Thus A440 can be represented alternatively by 440 (*cps*), 69 (*midinn*), 8.09 (*pch*), or 8.75 (*oct*). Microtonal divisions of the *pch* semitone can be encoded by using more than two decimal places.

The mnemonics of the pitch conversion units are derived from morphemes of the forms involved, the second morpheme describing the source and the first morpheme the object (result). Thus *cpspch*(8.09) will convert the pitch argument 8.09 to its *cps* (or Hertz) equivalent, giving the value of 440. Since the argument is constant over the duration of the note, this conversion will take place at *i*-time, before any samples for the current note are produced.

By contrast, the conversion *cpsoct*(8.75 + *k1*) which gives the value of A440 transposed by the octave interval *k1*. The calculation will be repeated every *k*-period since that is the rate at which *k1* varies.



### Note

The conversion from *pch*, *oct*, or *midinn* into *cps* is not a linear operation but involves an exponential process that could be time-consuming when executed repeatedly. Csound now

uses a built-in table lookup to do this efficiently, even at audio rates. Because the table index is truncated without interpolation, pitch resolution when using one of these opcodes is limited to 8192 discrete and equal divisions of the octave, and some pitches of the standard 12-tone equally-tempered scale are very slightly mistuned (by at most 0.15 cents).

If you need more precision in the calculation, use *cps2pch* or *cpsxpch* instead.

## Examples

Here is an example of the *cpspch* opcode. It uses the file *cpspch.csd* [examples/cpspch.csd].

### Example 162. Example of the *cpspch* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o cpspch.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; Convert pitch-class value into Hz

ipch = p4
icps = cpspch(ipch)
print icps
asig oscil 0.7, icps, 1
outs asig, asig

endin

</CsInstruments>
<CsScore>
;sine wave.
f 1 0 16384 10 1

i 1 0 1 8.01
i 1 + 1 8.02
i 1 + 1 8.03
i 1 + .5 5.09

e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
instr 1: icps = 277.167
instr 1: icps = 293.656
instr 1: icps = 311.101
instr 1: icps = 54.995
```

## See Also

*cps2pch, cpsoct, cpsxpch, octcps, octpch, pchoct, cpsmidinn, octmidinn, pchmidinn*

# cpstmid

cpstmid — Get a MIDI note number (allows customized micro-tuning scales).

## Description

This unit is similar to *cpsmidi*, but allows fully customized micro-tuning scales.

## Syntax

```
icps cpstmid ifn
```

## Initialization

*ifn* -- function table containing the parameters (*numgrades*, *interval*, *basefreq*, *basekeymidi*) and the tuning ratios.

## Performance

Init-rate only

*cpsmid* requires five parameters, the first, *ifn*, is the function table number of the tuning ratios, and the other parameters must be stored in the function table itself. The function table *ifn* should be generated by *GEN02*, with normalization inhibited. The first four values stored in this function are:

1. *numgrades* -- the number of grades of the micro-tuning scale
2. *interval* -- the frequency range covered before repeating the grade ratios, for example 2 for one octave, 1.5 for a fifth etc.
3. *basefreq* -- the base frequency of the scale in Hz
4. *basekeymidi* -- the MIDI note number to which *basefreq* is assigned unmodified

After these four values, the user can begin to insert the tuning ratios. For example, for a standard 12 note scale with the base frequency of 261 Hz assigned to the key number 60, the corresponding f-statement in the score to generate the table should be:

```
;      numgrades interval basefreq basekeymidi tuning ratios (equal temp)
f1 0 64 -2 12 2 261 60 1 1.059463094359 1.122462048309 1.189207115003 ..etc...
```

Another example with a 24 note scale with a base frequency of 440 assigned to the key number 48, and a repetition interval of 1.5:

```
;      numgrades interval basefreq basekeymidi tuning-ratios (equal temp)
f1 0 64 -2 24 1.5 440 48 1 1.01 1.02 1.03 ..etc...
```

## Examples

Here is an example of the cpstmid opcode. It uses the file *cpstmid.csd* [examples/cpstmid.csd].

**Example 163. Example of the cpstmidi opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      -MO    ;;RT audio I/O with MIDI in
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o cpstmidi.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
; after an example from Kevin Conder
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

; Table #1, a normal 12-tone equal temperament scale.
; numgrades = 12 (twelve tones)
; interval = 2 (one octave)
; basefreq = 261.659 (Middle C)
; basekeymidi = 60 (Middle C)
gitemp ftgen 1, 0, 64, -2, 12, 2, 261.659, 60, 1.00, \
        1.059, 1.122, 1.189, 1.260, 1.335, 1.414, \
        1.498, 1.588, 1.682, 1.782, 1.888, 2.000

instr 1

ifn = 1
icps cpstmidi ifn
print icps
asig oscil 0.6, icps, 2
outs asig, asig

endin

</CsInstruments>
<CsScore>
f 0 20
;sine wave.
f 2 0 16384 10 1

e

</CsScore>
</CsoundSynthesizer>

```

**See Also**

*cpsmidi*, *GEN02*

**Credits**

Author: Gabriel Maldonado  
Italy  
1998

New in Csound version 3.492

# cpstun

cpstun — Returns micro-tuning values at k-rate.

## Description

Returns micro-tuning values at k-rate.

## Syntax

```
kcps cpstun ktrig, kindex, kfn
```

## Performance

*kcps* -- Return value in cycles per second.

*ktrig* -- A trigger signal used to trigger the evaluation.

*kindex* -- An integer number denoting an index of scale.

*kfn* -- Function table containing the parameters (numgrades, interval, basefreq, basekeymidi) and the tuning ratios.

These opcodes are similar to *cpstmid*, but work without necessity of MIDI.

*cpstun* works at k-rate. It allows fully customized micro-tuning scales. It requires a function table number containing the tuning ratios, and some other parameters stored in the function table itself.

*kindex* arguments should be filled with integer numbers expressing the grade of given scale to be converted in cps. In *cpstun*, a new value is evaluated only when *ktrig* contains a non-zero value. The function table *kfn* should be generated by *GEN02* and the first four values stored in this function are parameters that express:

- numgrades -- The number of grades of the micro-tuning scale.
- interval -- The frequency range covered before repeating the grade ratios, for example 2 for one octave, 1.5 for a fifth etcetera.
- basefreq -- The base frequency of the scale in cycles per second.
- basekey -- The integer index of the scale to which to assign basefreq unmodified.

After these four values, the user can begin to insert the tuning ratios. For example, for a standard 12-grade scale with the base-frequency of 261 cps assigned to the key-number 60, the corresponding f-statement in the score to generate the table should be:

```
;          numgrades  basefreq  tuning-ratios (eq.temp) .....  
;          interval   basekey  
f1 0 64 -2 12      2      261    60    1    1.059463 1.12246 1.18920 ..etc...
```

Another example with a 24-grade scale with a base frequency of 440 assigned to the key-number 48, and a repetition interval of 1.5:

```

;          numgrades    basefreq    tuning-ratios .....
;          interval     basekey
f1 0 64 -2      24      1.5      440      48      1      1.01  1.02  1.03  ..etc...

```

## Examples

Here is an example of the `cpstun` opcode. It uses the file `cpstun.csd` [examples/cpstun.csd].

### Example 164. Example of the `cpstun` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o cpstun.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Table #1, a normal 12-tone equal temperament scale.
; numgrades = 12 (twelve tones)
; interval = 2 (one octave)
; basefreq = 261.659 (Middle C)
; basekeymidi = 60 (Middle C)
gitemp ftgen 1, 0, 64, -2, 12, 2, 261.659, 60, 1.00, \
          1.059, 1.122, 1.189, 1.260, 1.335, 1.414, \
          1.498, 1.588, 1.682, 1.782, 1.888, 2.000

; Instrument #1.
instr 1
; Set the trigger.
ktrig init 1

; Use Table #1.
kfn init 1

; If the base key (note #60) is C, then 9 notes
; above it (note #60 + 9 = note #69) should be A.
kindex init 69

k1 cpstun ktrig, kindex, kfn

printk2 k1
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
e

```

```
</CsScore>  
</CsoundSynthesizer>
```

Its output should include lines like this:

```
i1 440.11044
```

## See Also

*cpstmid*, *cpstuni*, *GEN02*

## Credits

Example written by Kevin Conder.

# cpstuni

cpstuni — Returns micro-tuning values at init-rate.

## Description

Returns micro-tuning values at init-rate.

## Syntax

```
icps cpstuni index, ifn
```

## Initialization

*icps* -- Return value in cycles per second.

*index* -- An integer number denoting an index of scale.

*ifn* -- Function table containing the parameters (numgrades, interval, basefreq, basekeymidi) and the tuning ratios.

## Performance

These opcodes are similar to *cpstmid*, but work without necessity of MIDI.

*cpstuni* works at init-rate. It allows fully customized micro-tuning scales. It requires a function table number containing the tuning ratios, and some other parameters stored in the function table itself.

The *index* argument should be filled with integer numbers expressing the grade of given scale to be converted in cps. The function table *ifn* should be generated by *GEN02* and the first four values stored in this function are parameters that express:

- numgrades -- The number of grades of the micro-tuning scale.
- interval -- The frequency range covered before repeating the grade ratios, for example 2 for one octave, 1.5 for a fifth etcetera.
- basefreq -- The base frequency of the scale in cycles per second.
- basekey -- The integer index of the scale to which to assign basefreq unmodified.

After these four values, the user can begin to insert the tuning ratios. For example, for a standard 12-grade scale with the base-frequency of 261 cps assigned to the key-number 60, the corresponding f-statement in the score to generate the table should be:

```
;          numgrades  basefreq  tuning-ratios (eq.temp) .....  
;          interval   basekey  
f1 0 64 -2 12      2    261    60    1    1.059463 1.12246 1.18920 ..etc...
```

Another example with a 24-grade scale with a base frequency of 440 assigned to the key-number 48, and a repetition interval of 1.5:

```

;                               numgrades    basefreq    tuning-ratios .....
;                               interval     basekey
f1 0 64 -2                      24         1.5         440         48         1         1.01 1.02 1.03 ..etc...

```

## Examples

Here is an example of the `cpstuni` opcode. It uses the file `cpstuni.csd` [examples/cpstuni.csd].

### Example 165. Example of the `cpstuni` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o cpstuni.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Table #1, a normal 12-tone equal temperament scale.
; numgrades = 12 (twelve tones)
; interval = 2 (one octave)
; basefreq = 261.659 (Middle C)
; basekeymidi = 60 (Middle C)
gitemp ftgen 1, 0, 64, -2, 12, 2, 261.659, 60, 1.00, \
        1.059, 1.122, 1.189, 1.260, 1.335, 1.414, \
        1.498, 1.588, 1.682, 1.782, 1.888, 2.000

; Instrument #1.
instr 1
; Use Table #1.
ifn = 1

; If the base key (note #60) is C, then 9 notes
; above it (note #60 + 9 = note #69) should be A.
index = 69

i1 cpstuni index, ifn

print i1
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
e

</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like this:

```
instr 1:  i1 = 440.110
```

## See Also

*cpstmid*, *cpstun*, *GEN02*

## Credits

Written by Gabriel Maldonado.

Example written by Kevin Conder.

# cpsxpch

`cpsxpch` — Converts a pitch-class value into cycles-per-second (Hz) for equal divisions of any interval.

## Description

Converts a pitch-class value into cycles-per-second (Hz) for equal divisions of any interval. There is a restriction of no more than 100 equal divisions.

## Syntax

```
icps cpsxpch ipch, iequal, irepeat, ibase
```

## Initialization

`ipch` -- Input number of the form 8ve.pc, indicating an 'octave' and which note in the octave.

`iequal` -- if positive, the number of equal intervals into which the 'octave' is divided. Must be less than or equal to 100. If negative, is the number of a table of frequency multipliers.

`irepeat` -- Number indicating the interval which is the 'octave.' The integer 2 corresponds to octave divisions, 3 to a twelfth, 4 is two octaves, and so on. This need not be an integer, but must be positive.

`ibase` -- The frequency which corresponds to pitch 0.0



### Note

1. The following are essentially the same

```
ia = cpspch(8.02)
ib  cps2pch 8.02, 12
ic  cpsxpch 8.02, 12, 2, 1.02197503906
```

2. These are opcodes not functions

3. Negative values of `ipch` are allowed, but not negative `irepeat`, `iequal` or `ibase`.

## Examples

Here is an example of the `cpsxpch` opcode. It uses the file `cpsxpch.csd` [examples/cpsxpch.csd].

### Example 166. Example of the `cpsxpch` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
```

```

; -o cpsxpch.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Use a normal twelve-tone scale.
ipch = 8.02
iequal = 12
irepeat = 2
ibase = 1.02197503906

icps cpsxpch ipch, iequal, irepeat, ibase

print icps
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
e

</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like this:

```
instr 1: icps = 293.666
```

Here is an example of the cpsxpch opcode using a 10.5 ET scale. It uses the file *cpsxpch\_105et.csd* [examples/cpsxpch\_105et.csd].

### Example 167. Example of the cpsxpch opcode using a 10.5 ET scale.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out Audio in
-odac -iadc ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o cpsxpch_105et.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Use a 10.5ET scale.
ipch = 4.02

```

```

iequal = 21
irepeat = 4
ibase = 16.35160062496

icps cpsxpch ipch, iequal, irepeat, ibase

print icps
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
e

</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like this:

```
instr 1: icps = 4776.824
```

Here is an example of the cpsxpch opcode using a Pierce scale centered on middle A. It uses the file *cpsxpch\_pierce.csd* [examples/cpsxpch\_pierce.csd].

### Example 168. Example of the cpsxpch opcode using a Pierce scale centered on middle A.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o cpsxpch_pierce.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Use a Pierce scale centered on middle A.
ipch = 2.02
iequal = 12
irepeat = 3
ibase = 261.62561

icps cpsxpch ipch, iequal, irepeat, ibase

print icps
endin

</CsInstruments>
<CsScore>

```

```
; Play Instrument #1 for one second.  
i 1 0 1  
e
```

```
</CsScore>  
</CsoundSynthesizer>
```

Its output should include lines like this:

```
instr 1: icps = 2827.762
```

## See Also

*cspch, cps2pch*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
1997

New in Csound version 3.492

# cpumeter

cpumeter — Reports the usage of cpu either total or per core.

## Description

Reports the usage of cpu either total or per core to monitor how close to max-out the processing is.

## Syntax

```
ktot[,kcpu1, kcpu2,...]cpumeter ifreq
```

## Initialization

*ifreq* is the time in seconds that the meter is refreshed. If this is too low then mainly figures of zero or one hundred occur. A value of 0.1 seems acceptable.

## Performance

*cpumeter* reads the total idle time in the last *ifreq* seconds and reports it as a percentage usage. If more than just *ktot* results are requested these report the same value for individual cores.

## Examples

Here is an example of the cpumeter opcode. It uses the file *cpumeter.csd* [examples/cpumeter.csd].

### Example 169. Example of the cpumeter opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o cpumeter.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

instr 1 ;cpu metering; stop when too large
k0 cpumeter 0.1
printk2 k0
if k0>70 then
event "i", 3, 0.1, 1
endif
endin
```

```
instr 2
  event_i    "i", 2, 1, 1000
  asig oscil 0.2, 440, 1
  out asig
endin

instr 3
  exitnow
endin
</CsInstruments>
<CsScore>
f 1 0 32768 10 1 ; sine wave

i 1 0 1000
i 2 0 1000
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*maxalloc, prealloc, cpuprc*

## Credits

Author: John ffitth  
May 2011

New in Csound version 5.14, for Linux/Unix and OSX only

# cpuprc

cpuprc — Control allocation of cpu resources on a per-instrument basis, to optimize realtime output.

## Description

Control allocation of cpu resources on a per-instrument basis, to optimize realtime output.

## Syntax

```
cpuprc insnum, ipercent
```

```
cpuprc Sinsname, ipercent
```

## Initialization

*insnum* -- instrument number or string

*Sinsname* -- instrument number or string

*ipercent* -- percent of cpu processing-time to assign. Can also be expressed as a fractional value.

## Performance

*cpuprc* sets the cpu processing-time percent usage of an instrument, in order to avoid buffer underrun in realtime performances, enabling a sort of polyphony threshold. The user must set *ipercent* value for each instrument to be activated in realtime. Assuming that the total theoretical processing time of the cpu of the computer is 100%, this percent value can only be defined empirically, because there are too many factors that contribute to limiting realtime polyphony in different computers.

For example, if *ipercent* is set to 5% for instrument 1, the maximum number of voices that can be allocated in realtime, is 20 ( $5\% * 20 = 100\%$ ). If the user attempts to play a further note while the 20 previous notes are still playing, Csound inhibits the allocation of that note and will display the following warning message:

```
can't allocate last note because it exceeds 100% of cpu time
```

In order to avoid audio buffer underruns, it is suggested to set the maximum number of voices slightly lower than the real processing power of the computer. Sometimes an instrument can require more processing time than normal. If, for example, the instrument contains an oscillator which reads a table that doesn't fit in cache memory, it will be slower than normal. In addition, any program running concurrently in multitasking, can subtract processing power to varying degrees.

At the start, all instruments are set to a default value of *ipercent* = 0.0% (i.e. zero processing time or rather infinite cpu processing-speed). This setting is OK for deferred-time sessions.

All instances of *cpuprc* must be defined in the header section, not in the instrument body.

## Examples

Here is an example of the *cpuprc* opcode. It uses the file *cpuprc.csd* [examples/cpuprc.csd].

### Example 170. Example of the cpuprc opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o cpuprc.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

cpuprc 1, 2
cpuprc 2, 30

instr 1 ;cpu processing-time percent usage is set to 2% for each note
asig oscil 0.2, 440, 1
outs asig, asig

endin

instr 2 ;cpu processing-time percent usage is set to 30% for each note
;so the 4 notes of the score exceeds 100% by far
asig oscil 0.2, 440, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 32768 10 1 ; sine wave

i 1 0 1
i 1 0 1
i 1 0 1
i 1 0 1

;too many notes to process,
;check Csound output!
i 2 3 1
i 2 3 1
i 2 3 1
i 2 3 1
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*maxalloc*, *prealloc*

## Credits

Author: Gabriel Maldonado

Italy  
July, 1999

New in Csound version 3.57; named instruments added version 5.13

# cross2

cross2 — Cross synthesis using FFT's.

## Description

This is an implementation of cross synthesis using FFT's.

## Syntax

```
ares cross2 ain1, ain2, isize, ioverlap, iwin, kbias
```

## Initialization

*isize* -- This is the size of the FFT to be performed. The larger the size the better the frequency response but a sloppy time response.

*ioverlap* -- This is the overlap factor of the FFT's, must be a power of two. The best settings are 2 and 4. A big overlap takes a long time to compile.

*iwin* -- This is the function table that contains the window to be used in the analysis. One can use the *GEN20* routine to create this window.

## Performance

*ain1* -- The stimulus sound. Must have high frequencies for best results.

*ain2* -- The modulating sound. Must have a moving frequency response (like speech) for best results.

*kbias* -- The amount of cross synthesis. 1 is the normal, 0 is no cross synthesis.

## Examples

Here is an example of the cross2 opcode. It uses the file *cross2.csd* [examples/cross2.csd] and *fox.wav* [examples/fox.wav].

### Example 171. Example of the cross2 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o cross2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
; after example from Kevin Conder
sr = 44100
ksmps = 32
```

```
nchnls = 2
0dbfs = 1

instr 1 ;play audio file

aout soundin "fox.wav"
outs aout, aout
endin

instr 2 ;cross-synthesize

icps = p4
ifn = p5 ; Use the "ahhc.aiff" sound and "eeec.aiff"
ain1 oscil 0.6, p4, ifn
ain2 soundin "fox.wav" ; Use the "fox.wav" as modulator

    ize = 4096
    ioverlap = 2
    iwin = 3
    kbias init 1

aout cross2 ain1, ain2, ize, ioverlap, iwin, kbias
outs aout, aout
endin

</CsInstruments>
<CsScore>
;audio files
f 1 0 128 1 "ahhc.aiff" 0 4 0
f 2 0 128 1 "eeec.aiff" 0 4 0

f 3 0 2048 20 2 ;windowing function

i 1 0 3

i 2 3 3 50 1 ;"eeec.aiff"
i 2 + 3 50 2 ;"ahhc.aiff"
i 2 + 3 100 1 ;"eeec.aiff"
i 2 + 3 100 2 ;"ahhc.aiff"
i 2 + 3 250 1 ;"eeec.aiff"
i 2 + 3 250 2 ;"ahhc.aiff"
i 2 + 3 20 1 ;"eeec.aiff"
i 2 + 3 20 2 ;"ahhc.aiff"
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1997

# crossfm

crossfm — Two mutually frequency and/or phase modulated oscillators.

## Description

Two oscillators, mutually frequency and/or phase modulated by each other.

## Syntax

```
a1, a2 crossfm xfrq1, xfrq2, xndx1, xndx2, kcps, ifn1, ifn2 [, iphs1] [, iphs2]
a1, a2 crossfmi xfrq1, xfrq2, xndx1, xndx2, kcps, ifn1, ifn2 [, iphs1] [, iphs2]
a1, a2 crosspm xfrq1, xfrq2, xndx1, xndx2, kcps, ifn1, ifn2 [, iphs1] [, iphs2]
a1, a2 crosspmi xfrq1, xfrq2, xndx1, xndx2, kcps, ifn1, ifn2 [, iphs1] [, iphs2]
a1, a2 crossfmpm xfrq1, xfrq2, xndx1, xndx2, kcps, ifn1, ifn2 [, iphs1] [, iphs2]
a1, a2 crossfmpmi xfrq1, xfrq2, xndx1, xndx2, kcps, ifn1, ifn2 [, iphs1] [, iphs2]
```

## Initialization

*ifn1* -- function table number for oscillator #1. Requires a wrap-around guard point.

*ifn2* -- function table number for oscillator #2. Requires a wrap-around guard point.

*iphs1* (optional, default=0) -- initial phase of waveform in table *ifn1*, expressed as a fraction of a cycle (0 to 1). A negative value will cause phase initialization to be skipped.

*iphs2* (optional, default=0) -- initial phase of waveform in table *ifn2*, expressed as a fraction of a cycle (0 to 1). A negative value will cause phase initialization to be skipped.

## Performance

*xfrq1* -- a factor that, when multiplied by the *kcps* parameter, gives the frequency of oscillator #1.

*xfrq2* -- a factor that, when multiplied by the *kcps* parameter, gives the frequency of oscillator #2.

*xndx1* -- the index of the modulation of oscillator #2 by oscillator #1.

*xndx2* -- the index of the modulation of oscillator #1 by oscillator #2.

*kcps* -- a common denominator, in cycles per second, for both oscillators frequencies.

*crossfm* implements a crossed frequency modulation algorithm. The audio-rate output of oscillator #1 is used to modulate the frequency input of oscillator #2 while the audio-rate output of oscillator #2 is used to modulate the frequency input of oscillator #1. This double feedback structure produces a rich set of sounds with some chaotic behaviour. *crossfmi* behaves like *crossfm* except that linear interpolation is used for table lookup.

*crosspm* and *crosspmi* implement cross phase modulation between two oscillators.

*crossfmpm* and *crossfmpmi* implement cross frequency/phase modulation between two oscillators. Oscillator #1 is frequency-modulated by oscillator #2 while oscillator #2 is phase-modulated by oscillator #1.

You can read my *paper* [<http://www.csoundjournal.com/issue12/crossfm.html>] in the Csound Journal for more information.



## Warning

Those opcodes may produce very rich spectra, especially with high modulation indexes, and in some cases foldover aliases may occur if the sampling rate is not high enough. Moreover the audio output may vary in function of the sampling rate, due to the non-linearity of the algorithm. In Csound, two other opcodes have this characteristic: *planet* and *chuap*.

## Examples

Here is an example of the *crossfm* opcode. It uses the file *crossfm.csd* [examples/crossfm.csd].

### Example 172. Example of the *crossfm* opcode.

```
<CsoundSynthesizer>

<CsOptions>
  -d -o dac
</CsOptions>

<CsInstruments>
  sr          =          96000
  ksmps       =          10
  nchnls      =          2
  0dbfs       =          1

  FLpanel "crossfmForm", 600, 400, 0, 0
  gkfrq1, ihfrq1 FLcount "Freq #1", 0, 20000, 0.001, 1, 1, 200, 30, 20, 50, -1
  gkfrq2, ihfrq2 FLcount "Freq #2", 0, 20000, 0.001, 1, 1, 200, 30, 20, 130, -1
  gkndx1, gkndx2, ihndx1, ihndx2 FLjoy "Indexes", 0, 10, 0, 10, 0, 0, -1, -1, 200, 200, 300, 50

  FLsetVal_i 164.5, ihfrq1
  FLsetVal_i 263.712, ihfrq2
  FLsetVal_i 1.5, ihndx1
  FLsetVal_i 3, ihndx2
  FLpanelEnd
  FLrun

  maxalloc 1, 2

  instr 1
  kamp      linen 0.5, 0.01, p3, 0.5
  a1,a2     crossfm gkfrq1, gkfrq2, gkndx1, gkndx2, 1, 1, 1
            outs   a1*kamp, a2*kamp
            endin
</CsInstruments>

<CsScore>
f1 0 16384 10 1 0
i1 0 60
e
</CsScore>
</CsoundSynthesizer>
```

In this example, an FLTK GUI is used to control in real-time the oscillators frequency with two *FLcount* widgets and the cross modulation indexes with one *FLjoy* widget. It uses a sampling rate of 96000Hz.

## See Also

More information on this opcode: <http://www.csoundjournal.com/issue12/crossfm.html> , written by François Pinot

## Credits

Author: François Pinot  
2005-2009

New in version 5.12

# crunch

crunch — Semi-physical model of a crunch sound.

## Description

*crunch* is a semi-physical model of a crunch sound. It is one of the PhISEM percussion opcodes. PhISEM (Physically Informed Stochastic Event Modeling) is an algorithmic approach for simulating collisions of multiple independent sound producing objects.

## Syntax

```
ares crunch iamp, idettack [, inum] [, idamp] [, imaxshake]
```

## Initialization

*iamp* -- Amplitude of output. Note: As these instruments are stochastic, this is only a approximation.

*idettack* -- period of time over which all sound is stopped

*inum* (optional) -- The number of beads, teeth, bells, timbrels, etc. If zero, the default value is 7.

*idamp* (optional) -- the damping factor, as part of this equation:

$$\text{damping\_amount} = 0.998 + (\text{idamp} * 0.002)$$

The default *damping\_amount* is 0.99806 which means that the default value of *idamp* is 0.03. The maximum *damping\_amount* is 1.0 (no damping). This means the maximum value for *idamp* is 1.0.

The recommended range for *idamp* is usually below 75% of the maximum value.

*imaxshake* (optional) -- amount of energy to add back into the system. The value should be in range 0 to 1.

## Examples

Here is an example of the crunch opcode. It uses the file *crunch.csd* [examples/crunch.csd].

### Example 173. Example of the crunch opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o crunch.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
```

```
ksmps = 32
nchnls = 2
Odbfs = 1

instr 1

asig crunch 0.8, 0.1, 7, p4
outs asig, asig

endin

</CsInstruments>
<CsScore>

i1 0 1 .9
i1 1 1 .1

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*cabasa, sandpaper, sekere, stix*

## Credits

Author: Perry Cook, part of the PhOLIES (Physically-Oriented Library of Imitated Environmental Sounds)  
Adapted by John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 4.07

Added notes by Rasmus Ekman on May 2002.

## ctrl14

ctrl14 — Allows a floating-point 14-bit MIDI signal scaled with a minimum and a maximum range.

### Description

Allows a floating-point 14-bit MIDI signal scaled with a minimum and a maximum range.

### Syntax

```
idest ctrl14 ichan, ictlno1, ictlno2, imin, imax [, ifn]
```

```
kdest ctrl14 ichan, ictlno1, ictlno2, kmin, kmax [, ifn]
```

### Initialization

*idest* -- output signal

*ichan* -- MIDI channel number (1-16)

*ictlno1* -- most-significant byte controller number (0-127)

*ictlno2* -- least-significant byte controller number (0-127)

*imin* -- user-defined minimum floating-point value of output

*imax* -- user-defined maximum floating-point value of output

*ifn* (optional) -- table to be read when indexing is required. Table must be normalized. Output is scaled according to *imax* and *imin* val.

### Performance

*kdest* -- output signal

*kmin* -- user-defined minimum floating-point value of output

*kmax* -- user-defined maximum floating-point value of output

*ctrl14* (i- and k-rate 14 bit MIDI control) allows a floating-point 14-bit MIDI signal scaled with a minimum and a maximum range. The minimum and maximum values can be varied at k-rate. It can use optional interpolated table indexing. It requires two MIDI controllers as input.

*ctrl14* differs from *midic14* because it can be included in score-oriented instruments without Csound crashes. It needs the additional parameter *ichan* containing the MIDI channel of the controller. MIDI channel is the same for all the controllers used in a single *ctrl14* opcode.

### Examples

Here is an example of the *ctrl14* opcode. It uses the file *ctrl14.csd* [examples/ctrl14.csd].

## Example 174. Example of the `ctrl14` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac  -M0  ;;RT audio I/O with MIDI in
;-iadc  ;;uncomment -iadc if RT audio input is needed too
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; expects MIDI controller input on channel 1
; run and move your midi controller to see result

imax = 1
imin = 0
ichan = 1
ictlno = 7

  initc14 1, ictlno, ictlno+1, 1 ; start at max. volume
  kamp ctrl14 ichan, ictlno, ictlno+1, imin, imax ; controller 7
  printk2 kamp
  asig oscil kamp, 220, 1
  outs asig, asig

endin

</CsInstruments>
<CsScore>
f 1 0 4096 10 1

i1 0 20

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*ctrl7, ctrl21, initc7, initc14, initc21, midic7, midic14, midic21*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# ctrl21

`ctrl21` — Allows a floating-point 21-bit MIDI signal scaled with a minimum and a maximum range.

## Description

Allows a floating-point 21-bit MIDI signal scaled with a minimum and a maximum range.

## Syntax

```
idest ctrl21 ichan, ictlno1, ictlno2, ictlno3, imin, imax [, ifn]
```

```
kdest ctrl21 ichan, ictlno1, ictlno2, ictlno3, kmin, kmax [, ifn]
```

## Initialization

*idest* -- output signal

*ichan* -- MIDI channel number (1-16)

*ictlno1* -- most-significant byte controller number (0-127)

*ictlno2* -- mid-significant byte controller number (0-127)

*ictlno3* -- least-significant byte controller number (0-127)

*imin* -- user-defined minimum floating-point value of output

*imax* -- user-defined maximum floating-point value of output

*ifn* (optional) -- table to be read when indexing is required. Table must be normalized. Output is scaled according to *imax* and *imin* val.

## Performance

*kdest* -- output signal

*kmin* -- user-defined minimum floating-point value of output

*kmax* -- user-defined maximum floating-point value of output

*ctrl21* (i- and k-rate 21 bit MIDI control) allows a floating-point 21-bit MIDI signal scaled with a minimum and a maximum range. Minimum and maximum values can be varied at k-rate. It can use optional interpolated table indexing. It requires three MIDI controllers as input.

*ctrl21* differs from *midic21* because it can be included in score oriented instruments without Csound crashes. It needs the additional parameter *ichan* containing the MIDI channel of the controller. MIDI channel is the same for all the controllers used in a single *ctrl21* opcode.

## Examples

Here is an example of the `ctrl21` opcode. It uses the file `ctrl21.csd` [examples/ctrl21.csd].

## Example 175. Example of the ctrl21 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac  -M0  ;;RT audio I/O with MIDI in
;-iadc  ;;uncomment -iadc if RT audio input is needed too
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; expects MIDI controller input on channel 1
; run and move your midi controller to see result

imax = 1
imin = 0
ichan = 1
ictlno = 7

  initc21 1, ictlno, ictlno+1, ictlno+2. 1  ; start at max. volume
  kamp ctrl21 ichan, ictlno, ictlno+1, ictlno+2, imin, imax ; controller 7
  printk2 kamp
  asig oscil kamp, 220, 1
  outs asig, asig

endin

</CsInstruments>
<CsScore>
f 1 0 4096 10 1

i1 0 20

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*ctrl7, ctrl14, initc7, initc14, initc21, midic7, midic14, midic21*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# ctrl7

*ctrl7* — Allows a floating-point 7-bit MIDI signal scaled with a minimum and a maximum range.

## Description

Allows a floating-point 7-bit MIDI signal scaled with a minimum and a maximum range.

## Syntax

```
idest ctrl7 ichan, ictlno, imin, imax [, ifn]
kdest ctrl7 ichan, ictlno, kmin, kmax [, ifn]
adest ctrl7 ichan, ictlno, kmin, kmax [, ifn] [, icutoff]
```

## Initialization

*idest* -- output signal

*ichan* -- MIDI channel (1-16)

*ictlno* -- MIDI controller number (0-127)

*imin* -- user-defined minimum floating-point value of output

*imax* -- user-defined maximum floating-point value of output

*ifn* (optional) -- table to be read when indexing is required. Table must be normalized. Output is scaled according to *imax* and *imin* val.

*icutoff* (optional) -- low pass filter cut-off frequency for smoothing a-rate output.

## Performance

*kdest*, *adest* -- output signal

*kmin* -- user-defined minimum floating-point value of output

*kmax* -- user-defined maximum floating-point value of output

*ctrl7* (i- and k-rate 7 bit MIDI control) allows a floating-point 7-bit MIDI signal scaled with a minimum and a maximum range. It also allows optional non-interpolated table indexing. Minimum and maximum values can be varied at k-rate.

*ctrl7* differs from *midic7* because it can be included in score-oriented instruments without Csound crashes. It also needs the additional parameter *ichan* containing the MIDI channel of the controller.

The a-rate version of *ctrl7* outputs an a-rate variable, which is low-pass filtered (smoothed). It contains an optional *icutoff* parameter, to set the cutoff frequency for the low-pass filter. The default is 5.

## Examples

Here is an example of the *ctrl7* opcode. It uses the file *ctrl7.csd* [examples/ctrl7.csd].

**Example 176. Example of the ctrl7 opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac  -M0  ;;RT audio I/O with MIDI in
;-iadc  ;;uncomment -iadc if RT audio input is needed too
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; expects MIDI controller input on channel 1
; run and move your midi controller to see result

imax = 1
imin = 0
ichan = 1
ictlno = 7

  initc7 1, 7, 1 ; start at max. volume
  kamp ctrl7 ichan, ictlno, imin, imax ; controller 7
  printk2 kamp
  asig oscil kamp, 220, 1
  outs asig, asig

endin

</CsInstruments>
<CsScore>
f 1 0 4096 10 1

i1 0 20

e
</CsScore>
</CsoundSynthesizer>

```

**Note**

ctrl7 only outputs k values once the controller is first moved. To set an initial k-value, a call to `initc7` is required. `Ctrlinit` does not work for this purpose.

**See Also**

*ctrl14, ctrl21, initc7, initc14, initc21, midic7, midic14, midic21*

**Credits**

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

The a-rate version of *ctrl7* was added in version 5.06

# ctrlinit

ctrlinit — Sets the initial values for a set of MIDI controllers.

## Description

Sets the initial values for a set of MIDI controllers.

## Syntax

```
ctrlinit ichnl, ictlno1, ival1 [, ictlno2] [, ival2] [, ictlno3] \  
[, ival3] [...ival32]
```

## Initialization

*ichnl* -- MIDI channel number (1-16)

*ictlno1*, *ictlno1*, etc. -- MIDI controller numbers (0-127)

*ival1*, *ival2*, etc. -- initial value for corresponding MIDI controller number

## Performance

Sets the initial values for a set of MIDI controllers.

## See Also

*massign*

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT, Cambridge, Mass.

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# cuserrnd

cuserrnd — Continuous USER-defined-distribution RaNDom generator.

## Description

Continuous USER-defined-distribution RaNDom generator.

## Syntax

```
aout cuserrnd kmin, kmax, ktableNum
```

```
iout cuserrnd imin, imax, itableNum
```

```
kout cuserrnd kmin, kmax, ktableNum
```

## Initialization

*imin* -- minimum range limit

*imax* -- maximum range limit

*itableNum* -- number of table containing the random-distribution function. Such table is generated by the user. See GEN40, GEN41, and GEN42. The table length does not need to be a power of 2

## Performance

*ktableNum* -- number of table containing the random-distribution function. Such table is generated by the user. See GEN40, GEN41, and GEN42. The table length does not need to be a power of 2

*kmin* -- minimum range limit

*kmax* -- maximum range limit

*cuserrnd* (continuous user-defined-distribution random generator) generates random values according to a continuous random distribution created by the user. In this case the shape of the distribution histogram can be drawn or generated by any GEN routine. The table containing the shape of such histogram must then be translated to a distribution function by means of GEN40 (see GEN40 for more details). Then such function must be assigned to the XtableNum argument of cuserrnd. The output range can then be rescaled according to the Xmin and Xmax arguments. cuserrnd linearly interpolates between table elements, so it is not recommended for discrete distributions (GEN41 and GEN42).

For a tutorial about random distribution histograms and functions see:

- D. Lorrain. "A panoply of stochastic cannons". In C. Roads, ed. 1989. Music machine. Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the cuserrnd opcode. It uses the file *cuserrnd.csd* [examples/cuserrnd.csd].

**Example 177. Example of the cuserrnd opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o cuserrnd.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; every run time same values

kuser cuserrnd 0, 100, 1
  printk .2, kuser
asig poscil .5, 220+kuser, 3
  outs asig, asig
endin

instr 2 ; every run time different values

  seed 0
kuser cuserrnd 0, 100, 1
  printk .2, kuser
asig poscil .5, 220+kuser, 3
  outs asig, asig
endin
</CsInstruments>
<CsScore>
f 1 0 16 -7 1 4 0 8 0 4 1 ;distrubution using GEN07
f 2 0 16384 40 1 ;GEN40 is to be used with cuserrnd
f 3 0 8192 10 1 ;sine

i 1 0 2
i 2 3 2
e
</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like these:

```

i 1 1 time      0.00067:    53.14918
i 1 1 time      0.20067:    0.00000
i 1 1 time      0.40067:    0.00000
i 1 1 time      0.60067:    96.80406
i 1 1 time      0.80067:    94.20729
i 1 1 time      1.00000:    0.00000
i 1 1 time      1.20067:    86.13032
i 1 1 time      1.40067:    31.37096
i 1 1 time      1.60067:    70.35889
i 1 1 time      1.80000:    0.00000
i 1 1 time      2.00000:    49.18914

```

WARNING: Seeding from current time 2006647442

i	2 time	3.00067:	21.45002
i	2 time	3.20067:	44.32333
i	2 time	3.40067:	46.05420
i	2 time	3.60000:	0.00000
i	2 time	3.80067:	41.32175
i	2 time	4.00000:	0.00000
i	2 time	4.20000:	63.72019
i	2 time	4.40067:	0.00000
i	2 time	4.60067:	0.00000
i	2 time	4.80067:	0.00000
i	2 time	5.00000:	74.49330

## See Also

*dusernd, urd*

## Credits

Author: Gabriel Maldonado

New in Version 4.16

# dam

dam — A dynamic compressor/expander.

## Description

This opcode dynamically modifies a gain value applied to the input sound *ain* by comparing its power level to a given threshold level. The signal will be compressed/expanded with different factors regarding that it is over or under the threshold.

## Syntax

```
ares dam asig, kthreshold, icompl, icompl2, irtime, iftime
```

## Initialization

*icompl* -- compression ratio for upper zone.

*icompl2* -- compression ratio for lower zone

*irtime* -- gain rise time in seconds. Time over which the gain factor is allowed to raise of one unit.

*iftime* -- gain fall time in seconds. Time over which the gain factor is allowed to decrease of one unit.

## Performance

*asig* -- input signal to be modified

*kthreshold* -- level of input signal which acts as the threshold. Can be changed at k-time (e.g. for ducking)

Note on the compression factors: A compression ratio of one leaves the sound unchanged. Setting the ratio to a value smaller than one will compress the signal (reduce its volume) while setting the ratio to a value greater than one will expand the signal (augment its volume).

## Examples

Because the results of the *dam* opcode can be subtle, I recommend looking at them in a graphical audio editor program like *audacity*. *audacity* is available for Linux, Windows, and the MacOS and may be downloaded from <http://audacity.sourceforge.net> [<http://audacity.sourceforge.net/>].

Here is an example of the *dam* opcode. It uses the file *dam.csd* [examples/dam.csd], and *beats.wav* [examples/beats.wav].

### Example 178. An example of the *dam* opcode compressing an audio signal.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform
```

```

-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o dam.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;normal audio

asig diskin2 "beats.wav", 1, 0, 1
outs asig, asig

endin

instr 2 ; compressed audio

kthreshold = 0.2
icompl = 0.8
icompp2 = 0.2
irtime = 0.01
iftime = 0.5
asig diskin2 "beats.wav", 1, 0, 1
asig dam asig, kthreshold, icompl, icompp2, irtime, iftime
outs asig, asig

endin

</CsInstruments>
<CsScore>

i 1 0 2
i 2 2.5 8.5

e
</CsScore>
</CsoundSynthesizer>

```

This example compresses the audio file “beats.wav”. You should hear a drum pattern repeat twice. The second time, the sound should be quieter (compressed) than the first.

Here is another example of the dam opcode. It uses the file *dam\_expanded.csd* [examples/dam\_expanded.csd], and *beats.wav* [examples/beats.wav].

### Example 179. An example of the dam opcode expanding an audio signal.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o dam_expanded.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

```

```
instr 1
  asig diskin2 "beats.wav", 1, 0, 1
    outs asig, asig
  endin

instr 2 ;expanded audio

  kthreshold = .5
  icompl = 2
  icomp2 = 3
  irtime = 0.01
  ifetime = 0.1

  asig diskin2 "beats.wav", 1, 0, 1
  asig dam asig, kthreshold, icompl, icomp2, irtime, ifetime
    outs asig*.2, asig*.2 ;adjust volume of expanded beat
  endin

</CsInstruments>
<CsScore>

i 1 0 2
i 2 2.5 6.5

e
</CsScore>
</CsSoundSynthesizer>
```

This example expands the audio file “beats.wav”. You should hear a drum pattern repeat twice. The second time, the sound should be louder (expanded) than the first. To prevent distortion the volume of the signal has been lowered.

## See Also

*compress*

## Credits

Author: Marc Resibois  
Belgium  
1997

New in version 3.47

# date

date — Returns the number seconds since a base date.

## Description

Returns the number seconds since a base date, using the operating system's clock. The base is 1 January 1970 for Csound using doubles, and 1 January 2010 for versions using floats. On operating systems with sufficient resolution the date includes fractional seconds.

## Syntax

```
ir[ , inano] date
```

```
kr[ , knano] date
```

## Initialization and Performance

*ir* -- value at i-time, of the system clock in seconds since the start of the epoch.

*kr* -- value at k-time, of the system clock in seconds since the start of the epoch.

*inano* -- value at i-time of the nanoseconds since last second tick.

*knano* -- value at k-time of the nanoseconds since last second tick.

Note that the base date was originally 1970 but since version 5.14 it has been changed as single precision floating point numbers are insufficient to indicate changes.

The optional answer and resolution including fractions of a second introduced in version 6.07.

## Examples

Here is an example of the date opcode. It uses the file *date.csd* [examples/date.csd].

### Example 180. Example of the date opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o date.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
instr 1
  ii,ij date
  print ii
  print ij
  Sa dates ii
  prints Sa
  Ss dates -1
  prints Ss
```

```
    St dates 1
    prints St
endin

</CsInstruments>

<CsScore>
i 1 0 1
e
</CsScore>

</CsoundSynthesizer>
```

Its output should include lines like this:

```
instr 1:  ii = 1447337826.778
instr 1:  ij = 778279830.000
Thu Nov 12 14:17:07 2015
Thu Nov 12 14:17:06 2015
Thu Jan  1 01:00:01 1970
```

## See Also

*dates*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
December 2006

New in Csound version 5.05

Changed in Csound version 5.14

# dates

dates — Returns as a string the date and time specified.

## Description

Returns as a string the date and time specified.

## Syntax

```
Sir dates [ itime]
```

## Initialization

*itime* -- the time in seconds since the start of the epoch. If omitted or negative the current time is taken.

*Sir* -- the date and time as a string.

## Examples

Here is an example of the dates opcode. It uses the file *dates.csd* [examples/dates.csd].

### Example 181. Example of the dates opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o dates.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

seed    0 ;each time different seed

instr 1
;;generating a different filename each time csound renders
itim     date
Stim     dates      itim
Syear    strsub     Stim, 20, 24
Smonth   strsub     Stim, 4, 7
Sday     strsub     Stim, 8, 10
iday     strtod     Sday
Shor     strsub     Stim, 11, 13
Smin     strsub     Stim, 14, 16
Ssec     strsub     Stim, 17, 19
Sfilnam  sprintf    "%s_%s_%02d_%s_%s_%s.wav", Syear, Smonth, iday, Shor,Smin, Ssec
;;rendering with random frequency, amp and pan, and writing to disk
ifreq    random     400, 1000
iamp     random     .1, 1
```

```
ipan    random    0, 1
asin    oscils    iamp, ifreq, 0
aL, aR  pan2         asin, ipan
        fout      Sfilnam, 14, aL, aR
        outs      aL, aR
        printf_i  "File '%s' written to the same directory as this CSD file is!\n", 1, Sfilnam

endin

</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
File '2011_Jan_05_19_14_46.wav' written to the same directory as this CSD file is!
Closing file '/home/user/csound/Output/2011_Jan_05_19_14_46.wav'...
```

## See Also

*date*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
December 2006

New in Csound version 5.05

# db

db — Returns the amplitude equivalent for a given decibel amount.

## Description

Returns the amplitude equivalent for a given decibel amount. This opcode is the same as *ampdb*.

## Syntax

db(x)

This function works at a-rate, i-rate, and k-rate.

## Initialization

x -- a value expressed in decibels.

## Performance

Returns the amplitude for a given decibel amount.

## Examples

Here is an example of the db opcode. It uses the file *db.csd* [examples/db.csd].

### Example 182. Example of the db opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o db.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

instr 1

idec = p4
iamp = db(idec)
print iamp
asig vco2 iamp, 110 ;sawtooth
outs asig, asig

endin
```

```
</CsInstruments>
<CsScore>

i 1 0 1 50
i 1 + 1 >
i 1 + 1 >
i 1 + 1 85
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like:

```
instr 1: iamp = 316.252
instr 1: iamp = 1211.582
instr 1: iamp = 4641.643
instr 1: iamp = 17782.420
```

## See Also

*ampdb, cent, octave, semitone*

New in version 4.16

# dbamp

dbamp — Returns the decibel equivalent of the raw amplitude  $x$ .

## Description

Returns the decibel equivalent of the raw amplitude  $x$ .

## Syntax

```
dbamp(x) (init-rate or control-rate args only)
```

## Examples

Here is an example of the dbamp opcode. It uses the file *dbamp.csd* [examples/dbamp.csd].

### Example 183. Example of the dbamp opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o dbamp.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

instr 1

iamp = p4
idb = dbamp(iamp)
print idb
asig vco2 iamp, 110 ;sawtooth
outs asig, asig

endin

</CsInstruments>
<CsScore>

i 1 0 1 100
i 1 + 1 1000
i 1 + 1 10000
i 1 + 1 20000
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
instr 1: idb = 40.000  
instr 1: idb = 60  
instr 1: idb = 80  
instr 1: idb = 86.021
```

## See Also

*ampdb, ampdbfs, dbfsamp*

# dbfsamp

dbfsamp — Returns the decibel equivalent of the raw amplitude  $x$ , relative to full scale amplitude.

## Description

Returns the decibel equivalent of the raw amplitude  $x$ , relative to full scale amplitude. Full scale is assumed to be 16 bit. New in Csound version 4.10.

## Syntax

`dbfsamp(x)` (init-rate or control-rate args only)

## Examples

Here is an example of the dbfsamp opcode. It uses the file `dbfsamp.csd` [examples/dbfsamp.csd].

### Example 184. Example of the dbfsamp opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o dbfsamp.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

instr 1

iamp = p4
idb = dbfsamp(iamp)
      print idb
asig vco2 iamp, 110 ;sawtooth
      outs asig, asig

endin

</CsInstruments>
<CsScore>

i 1 0 1 1
i 1 + 1 100
i 1 + 1 1000
i 1 + 1 10000
i 1 + 1 30000
e

</CsScore>
```

```
</CsoundSynthesizer>
```

Its output should include lines like this:

```
instr 1: idb = -90.309  
instr 1: idb = -50.309  
instr 1: idb = -30.309  
instr 1: idb = -10.309  
instr 1: idb = -0.767
```

## See Also

*ampdb*, *ampdbfs*, *dbamp*

# dcblock

dcblock — A DC blocking filter.

## Description

Implements the DC blocking filter

$$Y[i] = X[i] - X[i-1] + (\text{igain} * Y[i-1])$$

Based on work by Perry Cook.

## Syntax

```
ares dcblock ain [, igain]
```

## Initialization

*igain* -- the gain of the filter, which defaults to 0.99

## Performance

*ain* -- audio signal input



### Note

The new *dcblock2* opcode is an improved method of removing DC from an audio signal.

## Examples

The result can be viewed in a graphical audio editor program like *audacity*. *audacity* is available for Linux, Windows, and the MacOS and may be downloaded from <http://audacity.sourceforge.net> [<http://audacity.sourceforge.net/>].

Here is an example of the *dcblock* opcode. It uses the file *dcblock.csd* [examples/dcblock.csd], and *beat-s.wav* [examples/beats.wav].

### Example 185. Example of the *dcblock* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
;-o dcblock.wav -W ;; for file output any platform
```

```
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

instr 1 ;add DC to "beats.wav"

asig soundin "beats.wav"
asig = asig+5000 ;adds DC of 5000
outs asig, asig
endin

instr 2 ;dcblock audio

asig soundin "beats.wav"
asig = asig+5000 ;adds DC
adc dcblock asig ;remove DC again
outs adc, adc

endin

</CsInstruments>
<CsScore>

i 1 0 2
i 2 2 2
e

</CsScore>
</CsoundSynthesizer>
```

## See also

*dcblock2*

## Credits

Author: John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.49

February 2003: Thanks to a note from Anders Andersson, corrected the formula.

# dcblock2

dcblock2 — A DC blocking filter.

## Description

Implements a DC blocking filter with improved DC attenuation.

## Syntax

```
ares dcblock2 ain [, iorder] [, iskip]
```

## Initialization

*iorder* -- filter order, minimum 4th order, defaults to 128.

*iskip* -- set to 1 to skip initialization (defaults to 0).

## Performance

*ares* -- filtered audio signal

*ain* -- input audio signal



### Note

Using a value for *iorder* less than *ksmps* will not reduce DC offset efficiently.

## Examples

The result can be viewed in a graphical audio editor program like *audacity*. *audacity* is available for Linux, Windows, and the MacOS and may be downloaded from <http://audacity.sourceforge.net> [<http://audacity.sourceforge.net/>].

Here is an example of the *dcblock2* opcode. It uses the file *dcblock2.csd* [examples/dcblock2.csd], and *beats.wav* [examples/beats.wav].

### Example 186. Example of the *dcblock2* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
;-o dcblock2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```
sr = 44100
ksmps = 32
nchnls = 2

instr 1 ;add DC to "beats.wav"

asig soundin "beats.wav"
asig = asig+5000 ;adds DC of 5000
outs asig, asig
endin

instr 2 ;dcblock audio

asig soundin "beats.wav"
asig = asig+5000 ;adds DC
adc dcblock2 asig ;remove DC again
outs adc, adc

endin

</CsInstruments>
<CsScore>

i 1 0 2
i 2 2 2
e

</CsScore>
</CsoundSynthesizer>
```

## See also

*dcblock*

## Credits

By: Victor Lazzarini

New in Csound version 5.09

# dconv

dconv — A direct convolution opcode.

## Description

A direct convolution opcode.

## Syntax

```
ares dconv asig, isize, ifn
```

## Initialization

*isize* -- the size of the convolution buffer to use. If the buffer size is smaller than the size of *ifn*, then only the first *isize* values will be used from the table.

*ifn* -- table number of a stored function containing the impulse response for convolution.

## Performance

Rather than the analysis/resynthesis method of the *convolve* opcode, *dconv* uses direct convolution to create the result. For small tables it can do this quite efficiently, however larger table require much more time to run. *dconv* does (*isize* \* *ksmps*) multiplies on every k-cycle. Therefore, reverb and delay effects are best done with other opcodes (unless the times are short).

*dconv* was designed to be used with time varying tables to facilitate new realtime filtering capabilities.

## Examples

Here is an example of the dconv opcode. It uses the file *dconv.csd* [examples/dconv.csd].

### Example 187. Example of the dconv opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac        -iadc      -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o dconv.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

#define RANDI(A) #kout randi 1, kfq, $A*.001+iseed, 1
```

```
        tablew kout, $A, itable#

instr 1
itable  init  1
iseed   init  .6
isize   init  ftlen(itable)
kfq     line  1, p3, 10

$RANDI(0)
$RANDI(1)
$RANDI(2)
$RANDI(3)
$RANDI(4)
$RANDI(5)
$RANDI(6)
$RANDI(7)
$RANDI(8)
$RANDI(9)
$RANDI(10)
$RANDI(11)
$RANDI(12)
$RANDI(13)
$RANDI(14)
$RANDI(15)

asig    rand  10000, .5, 1
asig    butlp asig, 5000
asig    dconv asig, isize, itable

        out   asig *.5
endin

</CsInstruments>
<CsScore>

f1 0 16 10 1
i1 0 10
e

</CsScore>
</CsoundSynthesizer>
```

## See also

*pconvolve, convolve, fconv*

## Credits

Author: William "Pete" Moss  
2001

New in version 4.12

# dct

dct — Discrete Cosine Transform of a sample array (type-II DCT)

## Description

Applies a forward Discrete Cosine Transform to an input 1-dimensional array producing an array of the same size containing the transform. Currently only power-of-two sizes are implemented. Available in i-time or perf (k-rate) versions.

## Syntax

```
kout[] dct kin[]
```

```
iout[] dct iin[]
```

## Initialization

*iout[]* -- output array containing the DCT. It will be created if it does not exist.

*iin[]* -- input array containing the input

## Performance

*kout[]* -- output array containing the DCT. It will be created if it does not exist.

*kin[]* -- input array containing the input

## Examples

Here is an example of the dct opcode. It uses the file *dct.csd* [examples/dct.csd].

### Example 188. Example of the dct opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>

instr 1

iArr[] fillarray 1,2,3,4
iDCT[] dct iArr
iIDCT[] dctinv iDCT

prints "%.1f %.1f %.1f %.1f => %.1f %.1f %.1f %.1f\n",
    iArr[0], iArr[1], iArr[2], iArr[3],
    iIDCT[0], iIDCT[1], iIDCT[2], iIDCT[3]

endin
</CsInstruments>
```

```
<CsScore>  
i1 0 0  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

Vectorial opcodes, array opcodes, *dctinv*

## Credits

Author: Victor Lazzarini  
NUI Maynooth  
2014

New in version 6.04

# dctinv

dctinv — Inverse Discrete Cosine Transform of a sample array (type-III DCT)

## Description

Applies an Inverse Discrete Cosine Transform to an input 1-dimensional array producing an array of the same size containing the transform. Currently only power-of-two sizes are implemented. Available in i-time or perf (k-rate) versions.

## Syntax

```
kout[] dctinv kin[]  
iout[] dctinv iin[]
```

## Initialization

*iout[]* -- output array containing the IDCT. It will be created if it does not exist.

*iin[]* -- input array containing the input

## Performance

*kout[]* -- output array containing the IDCT. It will be created if it does not exist.

*kin[]* -- input array containing the input

## Examples

Here is an example of the `dctinv` opcode. It uses the file `dctinv.csd` [examples/dctinv.csd].

### Example 189. Example of the `dctinv` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
</CsOptions>  
<CsInstruments>  
  
instr 1  
  
iArr[] fillarray 1,2,3,4  
iDCT[] dct iArr  
iIDCT[] dctinv iDCT  
  
prints "%.1f %.1f %.1f %.1f => %.1f %.1f %.1f %.1f\n",  
        iArr[0], iArr[1], iArr[2], iArr[3],  
        iIDCT[0], iIDCT[1], iIDCT[2], iIDCT[3]  
  
endin  
</CsInstruments>
```

```
<CsScore>  
i1 0 0  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

Vectorial opcodes, array opcodes, *dct*

## Credits

Author: Victor Lazzarini  
NUI Maynooth  
2014

New in version 6.04

# delay

delay — Delays an input signal by some time interval.

## Description

A signal can be read from or written into a delay path, or it can be automatically delayed by some time interval.

## Syntax

```
ares delay asig, idlt [, iskip]
```

## Initialization

*idlt* -- requested delay time in seconds. This can be as large as available memory will permit. The space required for *n* seconds of delay is  $4n * sr$  bytes. It is allocated at the time the instrument is first initialized, and returned to the pool at the end of a score section.

*iskip* (optional, default=0) -- initial disposition of delay-loop data space (see *reson*). The default value is 0.

## Performance

*asig* -- audio signal

*delay* is a composite of *delayr* and *delayw*, both reading from and writing into its own storage area. It can thus accomplish signal time-shift, although modified feedback is not possible. There is no minimum delay period.

## Examples

Here is an example of the delay opcode. It uses the file *delay.csd* [examples/delay.csd].

### Example 190. Example of the delay opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o delay.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr      = 44100
ksmps  = 32
nchnls = 2
0dbfs  = 1
```

```
instr 1

adel init 0
ilev = p4 ;level of direct sound
idelay = p5 *.001 ;Delay in ms
ifd = p6 ;feedback

ain diskin2 "fox.wav", 1, 1
adel delay ain + (adel*ifd), idelay;ifd = amount of feedback
asig moogvcf adel, 1500, .6, 1 ;color feedback
outs asig*ilev, ain

endin

</CsInstruments>
<CsScore>
;Delay is in ms
i 1 0 15 2 200 .95 ;with feedback
i 1 4 5 2 20 .95
i 1 + 3 2 5 .95
i 1 + 3 3 5 0 ;no feedback

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*delayl, delayr, delayw*

# delay1

delay1 — Delays an input signal by one sample.

## Description

Delays an input signal by one sample.

## Syntax

```
ares delay1 asig [, iskip]
```

## Initialization

*iskip* (optional, default=0) -- initial disposition of delay-loop data space (see *reson*). The default value is 0.

## Performance

*delay1* is a special form of delay that serves to delay the audio signal *asig* by just one sample. It is thus functionally equivalent to the *delay* opcode but is more efficient in both time and space. This unit is particularly useful in the fabrication of generalized non-recursive filters.

## Examples

Here is an example of the delay and delay1 opcodes. It uses the file *delay1.csd* [examples/delay1.csd].

### Example 191. Example of the delay1 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; For Non-realtime ouput leave only the line below:
-o delay.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2
0dbfs = 1

instr 1
; Make white noise.
a0 random -1, 1

; Simple Lowpass filter
a1 delay1 a0
aout = (a0+0.99*a1)/2

; output white and filtered
```

```
        outs      aout, a0
    endin

    </CsInstruments>
    <CsScore>
    ; Play Instrument #1.
    i 1 0.0 3

    e

    </CsScore>
    </CsoundSynthesizer>
```

## See Also

*delay, delayr, delayw*

## Credits

Author: Barry Vercoe

Example written by John fitch.

# delayk

delayk — Delays an input signal by some time interval.

## Description

k-rate delay opcodes

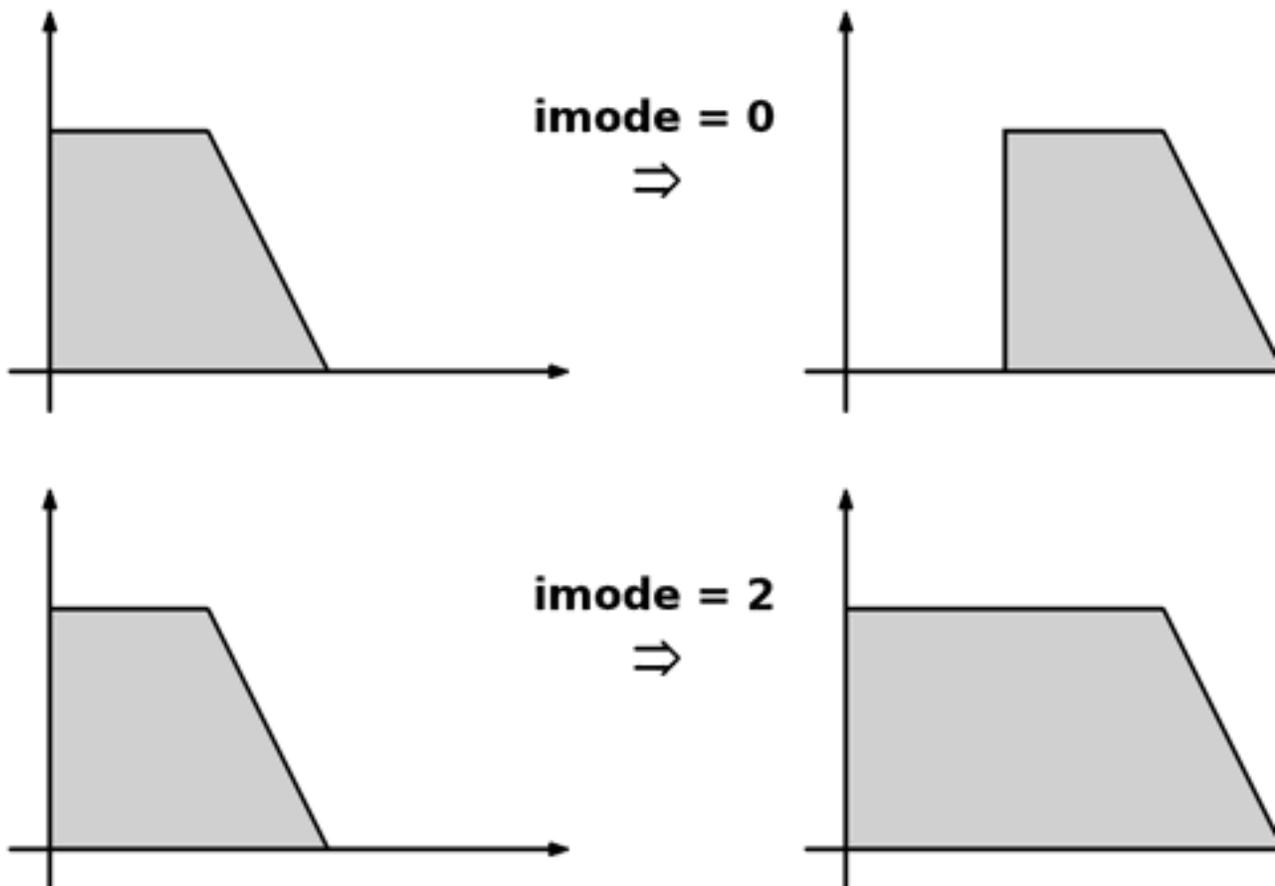
## Syntax

```
kr delayk   ksig, idel[, imode]
kr vdel_k   ksig, kdel, imdel[, imode]
```

## Initialization

*idel* -- delay time (in seconds) for delayk. It is rounded to the nearest integer multiple of a k-cycle (i.e.  $1/kr$ ).

*imode* -- sum of 1 for skipping initialization (e.g. in tied notes) and 2 for holding the first input value during the initial delay, instead of outputting zero. This is mainly of use when delaying envelopes that do not start at zero.



*imdel* -- maximum delay time for *vdel\_k*, in seconds.

## Performance

*kr* -- the output signal. Note: neither of the opcodes interpolate the output.

*ksig* -- the input signal.

*kdel* -- delay time (in seconds) for *vdel\_k*. It is rounded to the nearest integer multiple of a k-cycle (i.e. 1/kr).

## Examples

Here is an example of the delay opcode. It uses the file *delayk.csd* [examples/delayk.csd].

### Example 192. Example of the delayk opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o delayk.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
;example shows "delayk" for fm index and
;a second "delayk" for panning
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gisin ftgen 0, 0, 2^10, 10, 1

instr 1

kenv1 transeg 0, .02, 0, 1, 3.98, -6, 0 ;envelope
kenv2 delayk kenv1, 2 ;delayed by two seconds
kindx expon 5, p3, 1 ;fm index decreasing over p3
asig foscili .6, 400, 1, 11/4, kindx, gisin
kpan1 linseg 0, 4, 1 ;panning for first sound
kpan2 linseg 1, 4, 0 ;panning for second sound ...
kpan2 delayk kpan2, 2 ;delayed by two seconds
a1 = asig * kenv1
a2 = asig * kenv2

aL1,aR1 pan2 a1, kpan1
aL2,aR2 pan2 a2, kpan2
outs aL1+aL2, aR1+aR2

endin
</CsInstruments>
<CsScore>

i 1 0 6
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Istvan Varga.

# delayr

delayr — Reads from an automatically established digital delay line.

## Description

Reads from an automatically established digital delay line.

## Syntax

```
ares delayr idlt [, iskip]
```

## Initialization

*idlt* -- requested delay time in seconds. This can be as large as available memory will permit. The space required for *n* seconds of delay is  $4n * sr$  bytes. It is allocated at the time the instrument is first initialized, and returned to the pool at the end of a score section.

*iskip* (optional, default=0) -- initial disposition of delay-loop data space (see *reson*). The default value is 0.

## Performance

*delayr* reads from an automatically established digital delay line, in which the signal retrieved has been resident for *idlt* seconds. This unit must be paired with and precede an accompanying *delayw* unit. Any other Csound statements can intervene.

## Examples

Here is an example of the delayr opcode. It uses the file *delayr.csd* [examples/delayr.csd].

### Example 193. Example of the delayr opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o delayr.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gasig init 0
gidel = 1 ;delay time in seconds
```

```
instr 1

ain pluck .7, 440, 1000, 0, 1
  outs ain, ain

vincr gasig, ain ;send to global delay
endin

instr 2

ifedback = p4

abuf2 delayr gidel
adelL deltap .4 ;first tap (on left channel)
adelM deltap 1 ;second tap (on middle channel)
  delayw gasig + (adelL * ifedback)

abuf3 delayr gidel
kdel line 1, p3, .01 ;vary delay time
adelR deltap .65 * kdel ;one pitch changing tap (on the right chn.)
  delayw gasig + (adelR * ifedback)
;make a mix of all deayed signals
  outs adelL + adelM, adelR + adelM

clear gasig
endin

</CsInstruments>
<CsScore>

i 1 0 1
i 1 3 1
i 2 0 3 0 ;no feedback
i 2 3 8 .8 ;lots of feedback
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*delay, delayl, delayw*

# delayw

delayw — Writes the audio signal to a digital delay line.

## Description

Writes the audio signal to a digital delay line.

## Syntax

```
delayw asig
```

## Performance

*delayw* writes *asig* into the delay area established by the preceding *delayr* unit. Viewed as a pair, these two units permit the formation of modified feedback loops, etc. However, there is a lower bound on the value of *idlt*, which must be at least 1 control period (or  $1/kr$ ).

## Examples

Here is an example of the *delayw* opcode. It uses the file *delayw.csd* [examples/delayw.csd].

### Example 194. Example of the *delayw* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o delayw.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gasig  init 0
gidel  = 1 ;delay time in seconds

instr 1

ain  pluck .7, 440, 1000, 0, 1
     outs ain, ain

vincr gasig, ain ;send to global delay
endin

instr 2

ifedback = p4
```

```
abuf2 delayr gidel
adelL deltap .4 ;first tap (on left channel)
adelM deltap 1 ;second tap (on middle channel)
delayw gasig + (adelL * ifeedback)

abuf3 delayr gidel
kdel line 1, p3, .01 ;vary delay time
adelR deltap .65 * kdel ;one pitch changing tap (on the right chn.)
delayw gasig + (adelR * ifeedback)
;make a mix of all deayed signals
outs adelL + adelM, adelR + adelM

clear gasig
endin

</CsInstruments>
<CsScore>

i 1 0 1
i 1 3 1
i 2 0 3 0 ;no feedback
i 2 3 8 .8 ;lots of feedback
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*delay, delay1, delayr*

# deltap

`deltap` — Taps a delay line at variable offset times.

## Description

Tap a delay line at variable offset times.

## Syntax

```
ares deltap kdlt
```

## Performance

*kdlt* -- specifies the tapped delay time in seconds. Each can range from 1 control period to the full delay time of the read/write pair; however, since there is no internal check for adherence to this range, the user is wholly responsible. Each argument can be a constant, a variable, or a time-varying signal.

*deltap* extracts sound by reading the stored samples directly.

This opcode can tap into a *delayr/delayw* pair, extracting delayed audio from the *idlt* seconds of stored sound. There can be any number of *deltap* and/or *deltapi* units between a read/write pair. Each receives an audio tap with no change of original amplitude.

This opcode can provide multiple delay taps for arbitrary delay path and feedback networks. They can deliver either constant-time or time-varying taps, and are useful for building chorus effects, harmonizers, and Doppler shifts. Constant-time delay taps (and some slowly changing ones) do not need interpolated readout; they are well served by *deltap*. Medium-paced or fast varying dlt's, however, will need the extra services of *deltapi*.

*delayr/delayw* pairs may be interleaved. To associate a delay tap unit with a specific *delayr* unit, it not only has to be located between that *delayr* and the appropriate *delayw* unit, but must also precede any following *delayr* units. See Example 2. (This feature added in Csound version 3.57 by Jens Groh and John ffitich).

*N.B.* k-rate delay times are not internally interpolated, but rather lay down stepped time-shifts of audio samples; this will be found quite adequate for slowly changing tap times. For medium to fast-paced changes, however, one should provide a higher resolution audio-rate timeshift as input.

## Examples

### Example 195. `deltap` example #1

```
asource  buzz      1, 440, 20, 1
atime    linseg     1, p3/2,.01, p3/2,1 ; trace a distance in secs
ampfac   =          1/atime/atime      ; and calc an amp factor
adump    delayr     1                   ; set maximum distance
amove    deltapi    atime               ; move sound source past
          delayw     asource            ; the listener
          out        amove * ampfac
```

### Example 196. `deltap` example #2

```

ainput1 = .....
ainput2 = .....
kdlyt1  = .....
kdlyt2  = .....

;Read delayed signal, first delayr instance:
adump   delayr 4.0
adly1   deltap kdlyt1      ; associated with first delayr instance

;Read delayed signal, second delayr instance:
adump   delayr 4.0
adly2   deltap kdlyt2      ; associated with second delayr instance

;Do some cross-coupled manipulation:
afdbk1  =      0.7 * adly1 + 0.7 * adly2 + ainput1
afdbk2  =     -0.7 * adly1 + 0.7 * adly2 + ainput2

;Feed back signal, associated with first delayr instance:
        delayw afdbk1

;Feed back signal, associated with second delayr instance:
        delayw afdbk2
        outs   adly1, adly2

```

Here is yet another example of the `deltap` opcode. It uses the file `deltap.csd` [examples/deltap.csd].

### Example 197. Example of the `deltap` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o deltap.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gasig   init 0
gidel  = 1 ;delay time in seconds

instr 1

ain   pluck .7, 440, 1000, 0, 1
      outs ain, ain

vincr gasig, ain ;send to global delay
endin

instr 2

ifedback = p4

abuf2 delayr gidel
adelL  deltap .4 ;first tap (on left channel)
adelM  deltap 1 ;second tap (on middle channel)
delayw gasig + (adelL * ifedback)

```

```
abuf3 delayr gidel
kdel line 1, p3, .01 ;vary delay time
adelR deltap .65 * kdel ;one pitch changing tap (on the right chn.)
  delayw gasig + (adelR * ifeedback)
;make a mix of all deayed signals
outs adell + adelM, adelR + adelM

clear gasig
endin

</CsInstruments>
<CsScore>

i 1 0 1
i 1 3 1
i 2 0 3 0 ;no feedback
i 2 3 8 .8 ;lots of feedback
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*deltap3, deltapi, deltapn*

## deltap3

`deltap` — Taps a delay line at variable offset times, uses cubic interpolation.

### Description

Taps a delay line at variable offset times, uses cubic interpolation.

### Syntax

```
ares deltap3 xdl t
```

### Performance

`xdl t --` specifies the tapped delay time in seconds. Each can range from 1 control period to the full delay time of the read/write pair; however, since there is no internal check for adherence to this range, the user is wholly responsible. Each argument can be a constant, a variable, or a time-varying signal; the `xdl t` argument in `deltap3` implies that an audio-varying delay is permitted there.

`deltap3` is experimental, and uses cubic interpolation. (New in Csound version 3.50.)

This opcode can tap into a `delayr/delayw` pair, extracting delayed audio from the `idlt` seconds of stored sound. There can be any number of `deltap` and/or `deltapi` units between a read/write pair. Each receives an audio tap with no change of original amplitude.

This opcode can provide multiple delay taps for arbitrary delay path and feedback networks. They can deliver either constant-time or time-varying taps, and are useful for building chorus effects, harmonizers, and Doppler shifts. Constant-time delay taps (and some slowly changing ones) do not need interpolated readout; they are well served by `deltap`. Medium-paced or fast varying `dlt`'s, however, will need the extra services of `deltapi`.

`delayr/delayw` pairs may be interleaved. To associate a delay tap unit with a specific `delayr` unit, it not only has to be located between that `delayr` and the appropriate `delayw` unit, but must also precede any following `delayr` units. See Example 2. (This feature added in Csound version 3.57 by Jens Groh and John ffitch).

*N.B.* k-rate delay times are not internally interpolated, but rather lay down stepped time-shifts of audio samples; this will be found quite adequate for slowly changing tap times. For medium to fast-paced changes, however, one should provide a higher resolution audio-rate timeshift as input.

## Examples

### Example 198. `deltap` example #1

```
asource  buzz      1, 440, 20, 1
atime    linseg    1, p3/2,.01, p3/2,1 ; trace a distance in secs
ampfac   =         1/atime/atime      ; and calc an amp factor
adump    delayr    1                   ; set maximum distance
amove    deltapi   atime               ; move sound source past
         delayw    asource             ; the listener
out      amove * ampfac
```

### Example 199. `deltap` example #2

```

ainput1 = .....
ainput2 = .....
kdlyt1  = .....
kdlyt2  = .....

;Read delayed signal, first delayr instance:
adump   delayr 4.0
adly1   deltap kdlyt1      ; associated with first delayr instance

;Read delayed signal, second delayr instance:
adump   delayr 4.0
adly2   deltap kdlyt2      ; associated with second delayr instance

;Do some cross-coupled manipulation:
afdbk1  =      0.7 * adly1 + 0.7 * adly2 + ainput1
afdbk2  =     -0.7 * adly1 + 0.7 * adly2 + ainput2

;Feed back signal, associated with first delayr instance:
        delayw afdbk1

;Feed back signal, associated with second delayr instance:
        delayw afdbk2
        outs   adly1, adly2

```

Here is yet another example of the `deltap3` opcode. It uses the file `deltap3.csd` [examples/deltap3.csd].

### Example 200. Example of the `deltap3` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o deltap3.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gasig   init 0
gidel  = 1 ;delay time in seconds

instr 1

ain   pluck .7, 440, 1000, 0, 1
      outs ain, ain

vincr gasig, ain ;send to global delay
endin

instr 2

ifedback = p4

abuf2   delayr gidel
adelL   deltap3 .4 ;first tap (on left channel)
adelM   deltap3 1 ;second tap (on middle channel)
        delayw gasig + (adelL * ifedback)

```

```
abuf3 delayr gidel
kdel line 1, p3, .01 ;vary delay time
adelR deltap3 .65 * kdel ;one pitch changing tap (on the right chn.)
  delayw gasig + (adelR * ifeedback)
;make a mix of all deayed signals
outs adell + adelM, adelR + adelM

clear gasig
endin

</CsInstruments>
<CsScore>

i 1 0 1
i 1 3 1
i 2 0 3 0 ;no feedback
i 2 3 8 .8 ;lots of feedback
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*deltap, deltapi, deltapn*

# deltapi

deltapi — Taps a delay line at variable offset times, uses interpolation.

## Description

Taps a delay line at variable offset times, uses interpolation.

## Syntax

```
ares deltapi xdl
```

## Performance

*xdl* -- specifies the tapped delay time in seconds. Each can range from 1 control period to the full delay time of the read/write pair; however, since there is no internal check for adherence to this range, the user is wholly responsible. Each argument can be a constant, a variable, or a time-varying signal; the *xdl* argument in *deltapi* implies that an audio-varying delay is permitted there.

*deltapi* extracts sound by interpolated readout. By interpolating between adjacent stored samples *deltapi* represents a particular delay time with more accuracy, but it will take about twice as long to run.

This opcode can tap into a *delayr/delayw* pair, extracting delayed audio from the *idlt* seconds of stored sound. There can be any number of *deltap* and/or *deltapi* units between a read/write pair. Each receives an audio tap with no change of original amplitude.

This opcode can provide multiple delay taps for arbitrary delay path and feedback networks. They can deliver either constant-time or time-varying taps, and are useful for building chorus effects, harmonizers, and Doppler shifts. Constant-time delay taps (and some slowly changing ones) do not need interpolated readout; they are well served by *deltap*. Medium-paced or fast varying dlt's, however, will need the extra services of *deltapi*.

*delayr/delayw* pairs may be interleaved. To associate a delay tap unit with a specific *delayr* unit, it not only has to be located between that *delayr* and the appropriate *delayw* unit, but must also precede any following *delayr* units. See Example 2. (This feature added in Csound version 3.57 by Jens Groh and John fitch).

*N.B.* k-rate delay times are not internally interpolated, but rather lay down stepped time-shifts of audio samples; this will be found quite adequate for slowly changing tap times. For medium to fast-paced changes, however, one should provide a higher resolution audio-rate timeshift as input.

## Examples

### Example 201. *deltap* example #1

```
asource  buzz      1, 440, 20, 1
atime    linseg    1, p3/2, .01, p3/2, 1 ; trace a distance in secs
ampfac   =         1/atime/atime        ; and calc an amp factor
adump    delayr    1                    ; set maximum distance
amove    deltapi   atime                 ; move sound source past
          delayw    asource              ; the listener
out      amove * ampfac
```

**Example 202. deltapi example #2**

```

ainput1 = .....
ainput2 = .....
kdlyt1  = .....
kdlyt2  = .....

;Read delayed signal, first delayr instance:
adump   delayr 4.0
adly1   deltapi kdlyt1      ; associated with first delayr instance

;Read delayed signal, second delayr instance:
adump   delayr 4.0
adly2   deltapi kdlyt2      ; associated with second delayr instance

;Do some cross-coupled manipulation:
afdbk1  =      0.7 * adly1 + 0.7 * adly2 + ainput1
afdbk2  =     -0.7 * adly1 + 0.7 * adly2 + ainput2

;Feed back signal, associated with first delayr instance:
        delayw afdbk1

;Feed back signal, associated with second delayr instance:
        delayw afdbk2
        outs   adly1, adly2

```

Here is yet another example of the deltapi opcode. It uses the file *deltapi.csd* [examples/deltapi.csd].

**Example 203. Example of the deltapi opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsoundOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o deltapi.wav -W ;; for file output any platform
</CsoundOptions>
<CsoundInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gasig   init 0
gidel   = 1 ;delay time in seconds

instr 1

ain   pluck .7, 440, 1000, 0, 1
      outs ain, ain

vincr gasig, ain ;send to global delay
endin

instr 2

ifedback = p4

abuf2   delayr gidel

```

```
adell deltapi .4 ;first tap (on left channel)
adelM deltapi 1 ;second tap (on middle channel)
delayw gasig + (adell * ifeedback)

abuf3 delayr gidel
kdel line 1, p3, .01 ;vary delay time
adelR deltapi .65 * kdel ;one pitch changing tap (on the right chn.)
delayw gasig + (adelR * ifeedback)
;make a mix of all deayed signals
outs adell + adelM, adelR + adelM

clear gasig
endin

</CsInstruments>
<CsScore>

i 1 0 1
i 1 3 1
i 2 0 3 0 ;no feedback
i 2 3 8 .8 ;lots of feedback
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*deltap, deltap3, deltapn*

# deltapn

deltapn — Taps a delay line at variable offset times.

## Description

Tap a delay line at variable offset times.

## Syntax

```
ares deltapn xnumsamps
```

## Performance

*xnumsamps* -- specifies the tapped delay time in number of samples. Each can range from 1 control period to the full delay time of the read/write pair; however, since there is no internal check for adherence to this range, the user is wholly responsible. Each argument can be a constant, a variable, or a time-varying signal.

*deltapn* is identical to *deltapi*, except delay time is specified in number of samples, instead of seconds (Hans Mikelson).

This opcode can tap into a *delayr/delayw* pair, extracting delayed audio from the *idlt* seconds of stored sound. There can be any number of *deltap* and/or *deltapi* units between a read/write pair. Each receives an audio tap with no change of original amplitude.

This opcode can provide multiple delay taps for arbitrary delay path and feedback networks. They can deliver either constant-time or time-varying taps, and are useful for building chorus effects, harmonizers, and Doppler shifts. Constant-time delay taps (and some slowly changing ones) do not need interpolated readout; they are well served by *deltap*. Medium-paced or fast varying dlt's, however, will need the extra services of *deltapi*.

*delayr/delayw* pairs may be interleaved. To associate a delay tap unit with a specific *delayr* unit, it not only has to be located between that *delayr* and the appropriate *delayw* unit, but must also precede any following *delayr* units. See Example 2. (This feature added in Csound version 3.57 by Jens Groh and John ffitch).

*N.B.* k-rate delay times are not internally interpolated, but rather lay down stepped time-shifts of audio samples; this will be found quite adequate for slowly changing tap times. For medium to fast-paced changes, however, one should provide a higher resolution audio-rate timeshift as input.

## Examples

Here is an example of the *deltapn* opcode. It uses the file *deltapn.csd* [examples/deltapn.csd].

### Example 204. Example of the *deltapn* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform
```

```

-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o deltap3.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gasig  init 0
gidel = 1 ;delay time in seconds

instr 1

ain  pluck .7, 440, 1000, 0, 1
     outs ain, ain

vincr gasig, ain ;send to global delay
endin

instr 2

ifedback = p4

abuf2 delayr gidel
adelL  deltapn 4000 ;first tap (on left channel)
adelM  deltapn 44100 ;second tap (on middle channel)
       delayw gasig + (adelL * ifedback)

abuf3 delayr gidel
kdel  line 100, p3, 1 ;vary delay time
adelR  deltapn 100 * kdel ;one pitch changing tap (on the right chn.)
       delayw gasig + (adelR * ifedback)
;make a mix of all delayed signals
       outs adelL + adelM, adelR + adelM

clear gasig
endin

</CsInstruments>
<CsScore>

i 1 0 1
i 1 3 1
i 2 0 3 0 ;no feedback
i 2 3 8 .8 ;lots of feedback
e
</CsScore>
</CsSoundSynthesizer>

```

## See Also

*deltap, deltap3, deltapi*

# deltapx

`deltapx` — Read from or write to a delay line with interpolation.

## Description

`deltapx` is similar to `deltapi` or `deltap3`. However, it allows higher quality interpolation. This opcode can read from and write to a `delayr/delayw` delay line with interpolation.

## Syntax

```
aout deltapx adel, iwsiz
```

## Initialization

`iwsiz` -- interpolation window size in samples. Allowed values are integer multiplies of 4 in the range 4 to 1024. `iwsiz = 4` uses cubic interpolation. Increasing `iwsiz` improves sound quality at the expense of CPU usage, and minimum delay time.

## Performance

`aout` -- Output signal.

`adel` -- Delay time in seconds.

```
a1      delayr  idlr
        deltapxw a2, adl1, iws1
a3      deltapi adl2, iws2
        deltapxw a4, adl3, iws3
        delayw  a5
```

Minimum and maximum delay times:

```
idlr >= 1/kr                               Delay line length

adl1 >= (iws1/2)/sr                         Write before read
adl1 <= idlr - (1 + iws1/2)/sr              (allows shorter delays)

adl2 >= 1/kr + (iws2/2)/sr                  Read time
adl2 <= idlr - (1 + iws2/2)/sr
adl2 >= adl1 + (iws1 + iws2) / (2*sr)
adl2 >= 1/kr + adl3 + (iws2 + iws3) / (2*sr)

adl3 >= (iws3/2)/sr                         Write after read
adl3 <= idlr - (1 + iws3/2)/sr              (allows feedback)
```



### Note

Window sizes for opcodes other than `deltapx` are: `deltap`, `deltapn`: 1, `deltapi`: 2 (linear), `deltap3`: 4 (cubic)

## Examples

Here is an example of the `deltapx` opcode. It uses the file `deltapx.csd` [examples/deltapx.csd].

### Example 205. Example of the `deltapx` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o deltapx.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr      = 44100
ksmps   = 32
nchnls  = 2
0dbfs   = 1

instr 1

a1      phasor 300
a1      = a1 - 0.5
a_      delayr 1
adel    phasor 4
adel    = sin(2 * 3.14159265 * adel) * 0.01 + 0.2
deltapxw a1, adel, 32
adel    phasor 2
adel    = sin(2 * 3.14159265 * adel) * 0.01 + 0.2
deltapxw a1, adel, 32
adel    = 0.3
a2      deltapx adel, 32
a1      = 0
        delayw a1
        outs  a2*.7, a2*.7

endin
</CsInstruments>
<CsScore>

i1 0 5
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*deltapxw*

## Credits

Author: Istvan Varga  
August 2001

New in version 4.13

# deltapxw

`deltapxw` — Mixes the input signal to a delay line.

## Description

`deltapxw` mixes the input signal to a delay line. This opcode can be mixed with reading units (`deltap`, `deltapn`, `deltapi`, `deltap3`, and `deltapx`) in any order; the actual delay time is the difference of the read and write time. This opcode can read from and write to a `delayr/delayw` delay line with interpolation.

## Syntax

```
deltapxw ain, adel, iwsiz
```

## Initialization

`iwsiz` -- interpolation window size in samples. Allowed values are integer multiplies of 4 in the range 4 to 1024. `iwsiz = 4` uses cubic interpolation. Increasing `iwsiz` improves sound quality at the expense of CPU usage, and minimum delay time.

## Performance

`ain` -- Input signal.

`adel` -- Delay time in seconds.

```
a1    delayr  idlr
      deltapxw a2, adl1, iws1
a3    deltapx  adl2, iws2
      deltapxw a4, adl3, iws3
      delayw   a5
```

Minimum and maximum delay times:

```
idlr >= 1/kr                                Delay line length
adl1 >= (iws1/2)/sr                          Write before read
adl1 <= idlr - (1 + iws1/2)/sr                (allows shorter delays)
adl2 >= 1/kr + (iws2/2)/sr                    Read time
adl2 <= idlr - (1 + iws2/2)/sr
adl2 >= adl1 + (iws1 + iws2) / (2*sr)
adl2 >= 1/kr + adl3 + (iws2 + iws3) / (2*sr)
adl3 >= (iws3/2)/sr                          Write after read
adl3 <= idlr - (1 + iws3/2)/sr                (allows feedback)
```



### Note

Window sizes for opcodes other than `deltapx` are: `deltap`, `deltapn`: 1, `deltapi`: 2 (linear), `deltap3`: 4 (cubic)

## Examples

Here is an example of the `deltapxw` opcode. It uses the file `deltapxw.csd` [examples/deltapxw.csd].

### Example 206. Example of the `deltapxw` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o deltapxw.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr      = 44100
ksmps  = 32
nchnls = 2
0dbfs  = 1

instr 1

a1      phasor 300
a1      = a1 - 0.5
a_      delayr 1
adel    phasor 4
adel    = sin(2 * 3.14159265 * adel) * 0.01 + 0.2
deltapxw a1, adel, 32
adel    phasor 2
adel    = sin(2 * 3.14159265 * adel) * 0.01 + 0.2
deltapxw a1, adel, 32
adel    = 0.3
a2      deltapx adel, 32
a1      = 0
        delayw a1
        outs  a2*.7, a2*.7

endin
</CsInstruments>
<CsScore>

i1 0 5
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*deltapx*

## Credits

Author: Istvan Varga  
August 2001

New in version 4.13

# denorm

denorm — Mixes low level noise to a list of a-rate signals

## Description

Mixes low level ( $\sim 1e-20$  for floats, and  $\sim 1e-56$  for doubles) noise to a list of a-rate signals. Can be used before IIR filters and reverbs to avoid denormalized numbers which may otherwise result in significantly increased CPU usage.

## Syntax

```
denorm a1[ , a2[ , a3[ , ... ]]]
```

## Performance

*a1[ , a2[ , a3[ , ... ]]]* -- signals to mix noise with

Some processor architectures (particularly Pentium IVs) are very slow at processing extremely small numbers. These small numbers can appear as a result of some decaying feedback process like reverb and IIR filters. Low level noise can be added so that very small numbers are never reached, and they are 'absorbed' by this 'noise floor'.

If CPU usage goes to 100% at the end of reverb tails, or you get audio glitches in processes that shouldn't use too much CPU, using *denorm* before the culprits opcode or process might solve the problem.

## Examples

Here is an example of the denorm opcode. It uses the file *denorm.csd* [examples/denorm.csd].

### Example 207. Example of the denorm opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o denorm.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
; slightly simplified example from Istvan Varga 2006
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

garvb init 0

instr 1
```

```
a1 oscils 0.6, 440, 0
a2 linsegr 0, 0.005, 1, 3600, 1, 0.08, 0
a1 = a1 * a2
vincr garvb, a1
outs a1, a1
endin

instr 99 ;"Always on"

denorm garvb
aL, aR reverb garvb * 0.5, garvb * 0.5, 0.92, 10000
clear garvb
outs aL, aR
endin

</CsInstruments>
<CsScore>

i 99 0 -1 ;held by a negative p3, means "always on"
i 1 0 0.5
i 1 4 0.5
e 8 ;8 extra seconds after the performance

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Istvan Varga  
2005

# diff

diff — Modify a signal by differentiation.

## Description

Modify a signal by differentiation.

## Syntax

```
ares diff asig [, iskip]
kres diff ksig [, iskip]
```

## Initialization

*iskip* (optional) -- initial disposition of internal save space (see *reson*). The default value is 0.

## Performance

*integ* and *diff* perform integration and differentiation on an input control signal or audio signal. Each is the converse of the other, and applying both will reconstruct the original signal. Since these units are special cases of low-pass and high-pass filters, they produce a scaled (and phase shifted) output that is frequency-dependent. Thus *diff* of a sine produces a cosine, with amplitude  $2 * \pi * \text{Hz} / sr$  that of the original (for each component partial); *integ* will inversely affect the magnitudes of its component inputs. With this understanding, these units can provide useful signal modification.

## Examples

Here is an example of the diff opcode. It uses the file *diff.csd* [examples/diff.csd].

### Example 208. Example of the diff opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o diff.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

asig diskin2 "fox.wav", 1
```

```
        outs asig, asig

    endin

    instr 2 ; with diff

    asig diskin2 "fox.wav", 1
    ares diff asig
        outs ares, ares

    endin

    instr 3 ; with integ

    asig diskin2 "fox.wav", 1
    aint integ asig
    aint = aint*.05 ;way too loud
        outs aint, aint

    endin

    instr 4 ; with diff and integ

    asig diskin2 "fox.wav", 1
    ares diff asig
    aint integ ares
        outs aint, aint

    endin

</CsInstruments>
<CsScore>

i 1 0 1
i 2 1 1
i 3 2 1
i 4 3 1

e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*downsamp, integ, interp, samphold, upsamp*

# diode\_ladder

`diode_ladder` — Zero-delay feedback implementation of 4 pole diode ladder filter.

## Description

Zero-delay feedback implementation of a 4 pole (24 dB/oct) diode low-pass filter. This filter design was originally used in the EMS VCS3 and was the resonant filter in the Roland TB-303.

## Syntax

```
asig diode_ladder ain, xcf, xk [, inlp, isaturation, istor]
```

## Initialization

`istor` --initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

`asig` -- low-pass output signal.

`ain` -- input signal.

`xcf` -- filter cutoff frequency (i-, k-, or a-rate).

`xk` -- filter feedback value k (i-, k-, or a-rate) that controls resonance. Range 0.0-17.0. Self-oscillation occurs at 17.0.

`knlp` (optional, default=0) -- Non-linear processing method. 0 = no processing, 1 = higher-quality NLP, 2 = lower-quality NLP (faster). Method 1 uses  $(1.0 / \tanh(ksaturation)) * \tanh(ksaturation * input)$ . Method 2 uses  $\tanh(ksaturation * input)$ . Enabling NLP may increase the overall output of filter above unity and should be compensated for outside of the filter.

`ksaturation` (optional, default=1) -- saturation amount to use for non-linear processing. Values > 1 increase the steepness of the NLP curve.

## Examples

Here is an example of the `diode_ladder` opcode. It uses the file `diode_ladder.csd` [examples/diode\_ladder.csd].

### Example 209. Example of the `diode_ladder` opcode.

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
sr=44100
ksmps=16
nchnls=2
odbf=1
```

```

gi_sine ftgen 0, 0, 65537, 10, 1

gkcut init 6000

instr modulation
  gkcut = lfo(4000, 0.1) + 6000
endin

instr bass

  iamp = ampdbfs(-12)
  ipch = cps2pch(p4, 12)

  asig = vco2(0.5, ipch, 0)

  acut = expon:a(i(gkcut), p3, 200)
  aout = diode_ladder(asig, acut, 17, 1, 4)

  aout *= expseg:a(1.0, p3 - 0.05, 1.0, 0.05, 0.001)

  aout = limit(aout, -1.0, 1.0)

  outc(aout, aout)

endin

gipat[] init 8
gipat[0] = 6.00
gipat[1] = 7.00
gipat[2] = 6.00
gipat[3] = 7.00
gipat[4] = 5.07
gipat[5] = 6.07
gipat[6] = 5.08
gipat[7] = 6.08

instr player
  indx = p4

  ;; play instrument
  if(gipat[indx] > 0) then
    schedule("bass", 0, 0.2, gipat[indx])
  endif

  ;; temporal recursion
  schedule("player", 0.2, 0.1, (indx + 1) % lenarray(gipat))

endin

schedule("modulation", 0, -1)
schedule("player", 0, 0.1, 0)
event_i("e", 0, 0.1 * 128)

</CsInstruments>

<CsScore>
</CsScore>

</CsoundSynthesizer>

```

## References

This filter is based on the work of Will Pirkle that employs Vadim Zavalishin's work with bilinear transforms to create topology-preserving transform (TPT) implementations of analog filters.

1. Pirkle, Will. *Designing Software Synthesizer Plug-ins in C++: For RackAFX, VST3, and Audio Units*. CRC Press, 2014.
2. Pirkle, Will. *AN-6: Virtual Analog (VA) DiodeLadder Filter*. 2013.
3. Zavalishin, Vadim. "The Art of VA filter design." Native Instruments, 2012.

## Credits

Author: Steven Yi  
April 2017

New in Csound 6.09.0

# directory

directory — Reads a directory and outputs to a string array a list of file names.

## Description

Reads a directory for files and passes them to a string array. Users can set the file type by passing a file extension as a string.

## Syntax

```
SFiles[] directory SDirectory[, SExtention]
```

## Initialization

*SDirectory* -- a string that identifies the directory to browse for files

*SExtention* -- Optional. Sets the desired file type. If left out, all files names will be retrieved.

## Performance

*SFiles[]* -- a string array that holds the names of all files of a given type found in the directory.



### Note

This works at i-time only and will not pick up changes made to the directory after performance has started.

## Example

This example shows how `directory` can be used to browse for `.wav` files in a fixed location. `printf_i` is then used to print the file names. It uses the file `directory.csd` [examples/directory.csd].

### Example 210. Example of the `directory` opcode.

```
<CsoundSynthesizer>
<CsOptions>
-n
</CsOptions>
<CsInstruments>

;browse for text files in current directory
instr 1
iCnt init 0
SFileNames[] directory ".", ".txt"
iNumberOfFiles lenarray SFileNames

until iCnt>=iNumberOfFiles do
  printf_i "Filename = %s \n", 1, SFileNames[iCnt]
  iCnt = iCnt+1
od
endin
```

```
</CsInstruments>  
<CsScore>  
i1 0 1  
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Author: Rory Walsh  
2016

# diskgrain

diskgrain — Synchronous granular synthesis, using a soundfile as source.

## Description

*diskgrain* implements synchronous granular synthesis. The source sound for the grains is obtained by reading a soundfile containing the samples of the source waveform.

## Syntax

```
asig diskgrain Sfname, kamp, kfreq, kpitch, kgrsize, kprate, \  
      ifun, iolaps [,imaxgrsize , ioffset]
```

## Initialization

*Sfilename* -- source soundfile.

*ifun* -- grain envelope function table.

*iolaps* -- maximum number of overlaps,  $\max(kfreq) \cdot \max(kgrsize)$ . Estimating a large value should not affect performance, but exceeding this value will probably have disastrous consequences.

*imaxgrsize* -- max grain size in secs (default 1.0).

*ioffset* -- start offset in secs from beginning of file (default: 0).

## Performance

*kamp* -- amplitude scaling

*kfreq* -- frequency of grain generation, or density, in grains/sec.

*kpitch* -- grain pitch scaling (1=normal pitch, < 1 lower, > 1 higher; negative, backwards)

*kgrsize* -- grain size in secs.

*kprate* -- readout pointer rate, in grains. The value of 1 will advance the reading pointer 1 grain ahead in the source table. Larger values will time-compress and smaller values will time-expand the source signal. Negative values will cause the pointer to run backwards and zero will freeze it.

The grain generator has full control of frequency (grains/sec), overall amplitude, grain pitch (a sampling increment) and grain size (in secs), both as fixed or time-varying (signal) parameters. An extra parameter is the grain pointer speed (or rate), which controls which position the generator will start reading samples in the file for each successive grain. It is measured in fractions of grain size, so a value of 1 (the default) will make each successive grain read from where the previous grain should finish. A value of 0.5 will make the next grain start at the midway position from the previous grain start and finish, etc.. A value of 0 will make the generator read always from a fixed position (wherever the pointer was last at). A negative value will decrement pointer positions. This control gives extra flexibility for creating timescale modifications in the resynthesis.

*Diskgrain* will generate any number of parallel grain streams (which will depend on grain density/frequency), up to the *olaps* value (default 100). The number of streams (overlapped grains) is determined by

$\text{grainsize} * \text{grain\_freq}$ . More grain overlaps will demand more calculations and the synthesis might not run in realtime (depending on processor power).

*Diskgrain* can simulate FOF-like formant synthesis, provided that a suitable shape is used as grain envelope and a sinewave as the grain wave. For this use, grain sizes of around 0.04 secs can be used. The formant centre frequency is determined by the grain pitch. Since this is a sampling increment, in order to use a frequency in Hz, that value has to be scaled by  $\text{tablesize}/\text{sr}$ . Grain frequency will determine the fundamental.

This opcode is a variation on the *syncgrain* opcode.



### Note

diskgrain does not do any resampling, so if the sample rate of the source filename is not the same as csound's sr value there will be pitch shifts

## Examples

Here is an example of the diskgrain opcode. It uses the file *diskgrain.csd* [examples/diskgrain.csd].

### Example 211. Example of the diskgrain opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o diskgrain.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr      = 44100
ksmps  = 32
nchnls = 2
0dbfs  = 1

instr 1

iolaps = 2
igrsize = 0.04
ifreq  = iolaps/igrsize
ips    = 1/iolaps

istr = p4 /* timescale */
ipitch = p5 /* pitchscale */

a1 diskgrain "mary.wav", 1, ifreq, ipitch, igrsize, ips*istr, 1, iolaps
outs a1, a1

endin

</CsInstruments>
<CsScore>
f 1 0 8192 20 2 1 ;Hanning function

;          timescale  pitchscale
i 1 0 5 1 1
```

```
i 1 + 5 2 1
i 1 + 5 1 0.75
i 1 + 5 1.5 1.5
i 1 + 5 0.5 1.5

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
May 2007  
New in Csound 5.06

# diskin

diskin — Reads audio data from an external device or stream and can alter its pitch.

## Description

Reads audio data from an external device or stream and can alter its pitch.

## Syntax

```
ar1 [, ar2 [, ar3 [, ... arN]]] diskin ifilcod[, kpitch[, iskiptim \  
    [, iwraparound[, iformat[, iskipinit]]]]]
```

Note the N was 24 in versions before 5.14, and 40 after.

```
ar1[] diskin ifilcod[, kpitch[, iskiptim \  
    [, iwraparound[, iformat[, iskipinit]]]]]
```

(in this version, the number of output channels is not limited.)

## Initialization

*ifilcod* -- integer or character-string denoting the source soundfile name. An integer denotes the file soundin.filcod ; a character-string (in double quotes, spaces permitted) gives the filename itself, optionally a full pathname. If not a full path, the named file is sought first in the current directory, then in that given by the environment variable *SSDIR* (if defined) then by *SFDIR*. See also *GEN01*.

*iskiptim* (optional) -- time in seconds of input sound to be skipped. The default value is 0.

*iformat* (optional) -- specifies the audio data file format:

- 1 = 8-bit signed char (high-order 8 bits of a 16-bit integer)
- 2 = 8-bit A-law bytes
- 3 = 8-bit U-law bytes
- 4 = 16-bit short integers
- 5 = 32-bit long integers
- 6 = 32-bit floats
- 7 = 8-bit unsigned int (not available in Csound versions older than 5.00)
- 8 = 24-bit int (not available in Csound versions older than 5.00)
- 9 = 64-bit doubles (not available in Csound versions older than 5.00)

*iwraparound* -- 1 = on, 0 = off (wraps around to end of file either direction, enabling looping)

*iskipinit* switches off all initialisation if non zero (default =0). This was introduced in 4\_23f13 and csound5.

If *iformat* = 0 it is taken from the soundfile header, and if no header from the Csound *-o* command-line flag. The default value is 0.

## Performance

*a1 ... a24* -- output signals, in the range -0dbfs to 0dbfs. Any samples before the beginning (i.e. negative location) and after the end of the file are assumed to be zero, unless *iwrap* is non-zero. The number of output arguments must be the same as the number of sound file channels - which can be determined with the *filenchnls* opcode, otherwise an init error will occur.

*ar1[]* --- output signals, in a-rate array of size N, where N is the number of channels in the file. Arrays are automatically allocated to the correct size.

*kpitch* -- can be any real number. A negative number signifies backwards playback. The given number is a pitch ratio, where:

- *1* = normal pitch (default)
- *2* = 1 octave higher
- *3* = 12th higher, etc.
- *.5* = 1 octave lower
- *.25* = 2 octaves lower, etc.
- *-1* = normal pitch backwards
- *-2* = 1 octave higher backwards, etc.



### Note to Windows users

Windows users typically use back-slashes, “\”, when specifying the paths of their files. As an example, a Windows user might use the path “c:\music\samples\loop001.wav”. This is problematic because back-slashes are normally used to specify special characters.

To correctly specify this path in Csound, one may alternately:

- *Use forward slashes:* c:/music/samples/loop001.wav
- *Use back-slash special characters, “\\”:* c:\\music\\samples\\loop001.wav

## Examples

Here is an example of the *diskin* opcode. It uses the file *diskin.csd* [examples/diskin.csd], *beats.wav* [examples/beats.wav].

### Example 212. Example of the *diskin* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o diskin.wav -W ;; for file output any platform
```

```
</CsOptions>
<CsInstruments>

sr      = 44100
ksmps  = 32
nchnls = 2
odbfs  = 1

instr 1

ktrans linseg 1, 5, 2, 10, -2
a1      diskin "beats.wav", ktrans, 0, 1, 0, 32
outs   a1, a1

endin

</CsInstruments>
<CsScore>

i 1 0 15
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*in, inh, ino, inq, ins, soundin* and *diskin2*

## Credits

Authors: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

New in version 3.46

kpitch optional in version 6.06

Warning to Windows users added by Kevin Conder, April 2002

# diskin2

`diskin2` — Reads audio data from a file, and can alter its pitch using one of several available interpolation types, as well as convert the sample rate to match the orchestra `sr` setting.

## Description

Reads audio data from a file, and can alter its pitch using one of several available interpolation types, as well as convert the sample rate to match the orchestra `sr` setting. `diskin2` can also read multichannel files with any number of channels in the range 1 to 24 in versions before 5.14, and 40 after.

## Syntax

```
a1[, a2[, ... aN]] diskin2 ifilcod[, kpitch[, iskiptim \
    [, iwrap[, iformat[, iwsizel[, ibufsize[, iskipinit]]]]]]]
ar1[] diskin2 ifilcod[, kpitch[, iskiptim \
    [, iwrap[, iformat[, iwsizel[, ibufsize[, iskipinit]]]]]]]
```

(in the array output version, the number of output channels does not have an upper limit.)

## Initialization

*ifilcod* -- integer or character-string denoting the source soundfile name. An integer denotes the file `soundin.ifilcod`; a character-string (in double quotes, spaces permitted) gives the filename itself, optionally a full pathname. If not a full path, the named file is sought first in the current directory, then in those given by the environment variable `SSDIR` (if defined) then by `SFDIR`. See also `GEN01`. Note: files longer than  $2^{31}-1$  sample frames may not be played correctly on 32-bit platforms; this means a maximum length about 3 hours with a sample rate of 192000 Hz.

*iskiptim* (optional, defaults to zero) -- time in seconds of input sound to be skipped, assuming *kpitch*=1. Can be negative, to add *-iskiptim/kpitch* seconds of delay instead of skipping sound.



### Note

If *iwrap* is not 0 (locations are wrapped), *iskiptim* will not delay the sound if a negative value is used. Instead, the negative value will be "wrapped" from the end of the file.

*iwrap* (optional, defaults to zero) -- if set to any non-zero value, read locations that are negative or are beyond the end of the file are wrapped to the duration of the sound file instead of assuming zero samples. Useful for playing a file in a loop.



### Note

If *iwrap* is enabled, the file length should not be shorter than the interpolation window size (see below), otherwise there may be clicks in the sound output.

*iformat* (optional, defaults to zero) -- sample format, for raw (headerless) files only. This parameter is ignored if the file has a header. Allowed values are:

- 0: 16-bit short integers
- 1: 8-bit signed char (high-order 8 bits of a 16-bit integer)

- 2: 8-bit A-law bytes
- 3: 8-bit U-law bytes
- 4: 16-bit short integers
- 5: 32-bit long integers
- 6: 32-bit floats
- 7: 8-bit unsigned int
- 8: 24-bit int
- 9: 64-bit doubles

*iwsiz*e (optional, defaults to zero) -- interpolation window size, in samples. Can be one of the following:

- 1: round to nearest sample (no interpolation, for *kpitch*=1)
- 2: linear interpolation
- 4: cubic interpolation
- >= 8: *iwsiz*e point sinc interpolation with anti-aliasing (slow)

Zero or negative values select the default, which is cubic interpolation.



### Note

If interpolation is used, *kpitch* is automatically scaled by the ratio of the sample rate of the sound file and the orchestra, so that the file will always be played at the original pitch if *kpitch* is 1. However, the sample rate conversion is disabled if *iwsiz*e is 1.

*ibufsize* (optional, defaults to 0) -- buffer size in mono samples (not sample frames). This is only the suggested value, the actual setting will be rounded so that the number of sample frames is an integer power of two and is in the range 128 (or *iwsiz*e if greater than 128) to 1048576. The default, which is 4096, and is enabled by zero or negative values, should be suitable for most uses, but for non-realtime mixing of many large sound files, a high buffer setting is recommended to improve the efficiency of disk reads. For real time audio output, reading the files from a fast RAM file system (on platforms where this option is available) with a small buffer size may be preferred.

*iskipinit* (optional, defaults to 0) -- skip initialization if set to any non-zero value.

## Performance

*a1* ... *a24* -- output signals, in the range -0dbfs to 0dbfs. Any samples before the beginning (i.e. negative location) and after the end of the file are assumed to be zero, unless *iwrap* is non-zero. The number of output arguments must be the same as the number of sound file channels - which can be determined with the *filenchnls* opcode, otherwise an init error will occur.

*ar1*[] --- output signals, in a-rate array of size N, where N is the number of channels in the file. Arrays are automatically allocated to the correct size.



### Note

It is more efficient to read a single file with many channels, than many files with only a single channel, especially with high *iwsiz*e settings.

*kpitch* -- transpose the pitch of input sound by this factor (e.g. 0.5 means one octave lower, 2 is one octave higher, and 1 is the original pitch, which is the default value). Fractional and negative values are allowed (the latter results in playing the file backwards, however, in this case the skip time parameter should be set to some positive value, e.g. the length of the file, or *iwrap* should be non-zero, otherwise nothing would be played). If interpolation is enabled, and the sample rate of the file differs from the orchestra sample rate, the transpose ratio is automatically adjusted to make sure that *kpitch*=1 plays at the original pitch. Using a high *iwsize* setting (40 or more) can significantly improve sound quality when transposing up, although at the expense of high CPU usage.

## Examples

Here is an example of the *diskin2* opcode. It uses the file *diskin2.csd* [examples/diskin2.csd], *beats.wav* [examples/beats.wav].

### Example 213. Example of the *diskin2* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o diskin2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr        = 44100
ksmps    = 32
nchnls   = 2
0dbfs    = 1

instr 1

ktrans linseg 1, 5, 2, 10, -2
a1      diskin2 "beats.wav", ktrans, 0, 1, 0, 32
outs a1, a1

endin

</CsInstruments>
<CsScore>

i 1 0 15
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*in*, *inh*, *ino*, *inq*, *ins*, *soundin* and *diskin*

## Credits

Author: Istvan Varga  
2005

New in version 5.00

kpitch optional in version 6.06

# dispfft

displayfft — Displays the Fourier Transform of an audio or control signal.

## Description

These units will print orchestra init-values, or produce graphic display of orchestra control signals and audio signals. Uses X11 windows if enabled, else (or if `-g` flag is set) displays are approximated in ASCII characters.

## Syntax

```
dispfft xsig, iprd, iwsiz [, iwtyp] [, idbout] [, iwtflg] [,imin] [,imax]
```

## Initialization

*iprd* -- the period of display in seconds.

*iwsiz* -- size of the input window in samples. A window of *iwsiz* points will produce a Fourier transform of *iwsiz*/2 points, spread linearly in frequency from 0 to *sr*/2. *iwsiz* must be a power of 2, with a minimum of 16 and a maximum of 4096. The windows are permitted to overlap.

*iwtyp* (optional, default=0) -- window type. 0 = rectangular, 1 = Hanning. The default value is 0 (rectangular).

*idbout* (optional, default=0) -- units of output for the Fourier coefficients. 0 = magnitude, 1 = decibels. The default is 0 (magnitude).

*iwtflg* (optional, default=0) -- wait flag. If non-zero, each display is held until released by the user. The default value is 0 (no wait).

*imin* (optional, default=0) -- minimum FFT bin used in display.

*imax* (optional, default=winsize/2) -- maximum FFT bin to be used in display.

## Performance

*dispfft* -- displays the Fourier Transform of an audio or control signal (*asig* or *ksig*) every *iprd* seconds using the Fast Fourier Transform method.

## Examples

Here is an example of the `dispfft` opcode. It uses the file `dispfft.csd` [examples/dispfft.csd].

### Example 214. Example of the `dispfft` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>
```

```
; Select audio/midi flags here according to platform
-odac      ;;;realtime audio out
;-iadc     ;;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o dispfft.wav -W ;;; for file output any platform
</CsOptions>;be sure to NOT have -d in the CsOptions...
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kcps = 110
ifn = 1

knh line p4, p3, p5
asig buzz 1, kcps, knh, ifn
outs asig, asig

dispfft asig, .1, 2048, 0, 1

endin
</CsInstruments>
<CsScore>
;sine wave.
f 1 0 16384 10 1

i 1 0 3 20 20
i 1 + 3 3 3
i 1 + 3 150 1
e

</CsScore>
</CsSoundSynthesizer>
```

## See Also

*display, print*

# display

`display` — Displays the audio or control signals as an amplitude vs. time graph.

## Description

These units will print orchestra init-values, or produce graphic display of orchestra control signals and audio signals. Uses X11 windows if enabled, else (or if `-g` flag is set) displays are approximated in ASCII characters.

## Syntax

```
display xsig, iprd [, inprds] [, iwtflg]
```

## Initialization

`iprd` -- the period of display in seconds.

`inprds` (optional, default=1) -- Number of display periods retained in each display graph. A value of 2 or more will provide a larger perspective of the signal motion. The default value is 1 (each graph completely new). `inprds` is a scaling factor for the displayed waveform, controlling how many `iprd`-sized frames of samples are drawn in the window (the default and minimum value is 1.0). Higher `inprds` values are slower to draw (more points to draw) but will show the waveform scrolling through the window, which is useful with low `iprd` values.

`iwtflg` (optional, default=0) -- wait flag. If non-zero, each display is held until released by the user. The default value is 0 (no wait).

## Performance

`display` -- displays the audio or control signal `xsig` every `iprd` seconds, as an amplitude vs. time graph.

## Examples

Here is an example of the `display` opcode. It uses the file `display.csd` [examples/display.csd].

### Example 215. Example of the `display` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o display.wav -W ;; for file output any platform
</CsOptions>;be sure to NOT have -d in the CsOptions...
<CsInstruments>

sr = 44100
```

```
ksmps = 32
nchnls = 2
Odbfs = 1

instr 1

kcps = 110
ifn = 1

knh line p4, p3, p5
asig buzz 1, kcps, knh, ifn
outs asig, asig

display asig, .1, 3

endin
</CsInstruments>
<CsScore>
; sine wave.
f 1 0 16384 10 1

i 1 0 3 20 20
i 1 + 3 3 3
i 1 + 3 150 1
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*dispf*, *print*

## Credits

Comments about the *inprds* parameter contributed by Rasmus Ekman.

# distort

distort — Distort an audio signal via waveshaping and optional clipping.

## Description

Distort an audio signal via waveshaping and optional clipping.

## Syntax

```
ar distort asig, kdist, ifn[, ihp, istor]
```

## Initialization

*ifn* -- table number of a waveshaping function with extended guard point. The function can be of any shape, but it should pass through 0 with positive slope at the table mid-point. The table size need not be large, since it is read with interpolation.

*ihp* -- (optional) half-power point (in cps) of an internal low-pass filter. The default value is 10.

*istor* -- (optional) initial disposition of internal data space (see *reson*). The default value is 0.

## Performance

*asig* -- Audio signal to be processed

*kdist* -- Amount of distortion (usually between 0 and 1)

This unit distorts an incoming signal using a waveshaping function *ifn* and a distortion index *kdist*. The input signal is first compressed using a running rms, then passed through a waveshaping function which may modify its shape and spectrum. Finally it is rescaled to approximately its original power.

The amount of distortion depends on the nature of the shaping function and on the value of *kdist*, which generally ranges from 0 to 1. For low values of *kdist*, we should like the shaping function to pass the signal almost unchanged. This will be the case if, at the mid-point of the table, the shaping function is near-linear and is passing through 0 with positive slope. A line function from -1 to +1 will satisfy this requirement; so too will a sigmoid (sinusoid from 270 to 90 degrees). As *kdist* is increased, the compressed signal is expanded to encounter more and more of the shaping function, and if this becomes non-linear the signal is increasingly *bent* on read-through to cause distortion.

When *kdist* becomes large enough, the read-through process will eventually hit the outer limits of the table. The table is not read with wrap-around, but will “stick” at the end-points as the incoming signal exceeds them; this introduces clipping, an additional form of signal distortion. The point at which clipping begins will depend on the complexity (rms-to-peak value) of the input signal. For a pure sinusoid, clipping will begin only as *kdist* exceeds 0.7; for a more complex input, clipping might begin at a *kdist* of 0.5 or much less. *kdist* can exceed the clip point by any amount, and may be greater than 1.

The shaping function can be made arbitrarily complex for extra effect. It should generally be continuous, though this is not a requirement. It should also be well-behaved near the mid-point, and roughly balanced positive-negative overall, else some excessive DC offset may result. The user might experiment with more aggressive functions to suit the purpose. A generally positive slope allows the distorted signal to be mixed with the source without phase cancellation.

*distort* is useful as an effects process, and is usually combined with reverb and chorusing on effects busses. However, it can alternatively be used to good effect within a single instrument.

## Examples

Here is an example of the distort opcode. It uses the file *distort.csd* [examples/distort.csd].

### Example 216. Example of the distort opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o distort.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr      = 44100
ksmps  = 32
nchnls = 2
0dbfs  = 1

gifn ftgen 0,0, 257, 9, .5,1,270 ; define a sigmoid, or better
;gifn ftgen 0,0, 257, 9, .5,1,270,1.5,.33,90,2.5,.2,270,3.5,.143,90,4.5,.111,270

instr 1

kdist line 0, p3, 2 ; and over 10 seconds
asig poscil 0.3, 440, 1
out distort asig, kdist, gifn ; gradually increase the distortion
outs out, out

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1
i 1 0 10
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Written by Barry L. Vercoe for Extended Csound and released in Csound5.

# distort1

distort1 — Modified hyperbolic tangent distortion.

## Description

Implementation of modified hyperbolic tangent distortion. *distort1* can be used to generate wave shaping distortion based on a modification of the *tanh* function.

$$\text{aout} = \frac{\exp(\text{asig} * (\text{shape1} + \text{pregain})) - \exp(\text{asig} * (\text{shape2} - \text{pregain}))}{\exp(\text{asig} * \text{pregain}) + \exp(-\text{asig} * \text{pregain})}$$

## Syntax

```
ares distort1 asig, kpregain, kpostgain, kshape1, kshape2[, imode]
```

## Initialization

*imode* (Csound version 5.00 and later only; optional, defaults to 0) -- scales *kpregain*, *kpostgain*, *kshape1*, and *kshape2* for use with audio signals in the range -32768 to 32768 (*imode*=0), -0dbfs to 0dbfs (*imode*=1), or disables scaling of *kpregain* and *kpostgain* and scales *kshape1* by *kpregain* and *kshape2* by *-kpregain* (*imode*=2).

## Performance

*asig* -- is the input signal.

*kpregain* -- determines the amount of gain applied to the signal before waveshaping. A value of 1 gives slight distortion.

*kpostgain* -- determines the amount of gain applied to the signal after waveshaping.

*kshape1* -- determines the shape of the positive part of the curve. A value of 0 gives a flat clip, small positive values give sloped shaping.

*kshape2* -- determines the shape of the negative part of the curve.

## Examples

Here is an example of the *distort1* opcode. It uses the file *distort1.csd* [examples/distort1.csd].

### Example 217. Example of the distort1 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>
```

```

; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o distort1.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gadist init 0

instr 1
  iamp = p4
  ifqc = cpspch(p5)
  asig pluck iamp, ifqc, ifqc, 0, 1
  gadist = gadist + asig
endin

instr 50
  kpre init p4
  kpost init p5
  kshap1 init p6
  kshap2 init p7
  aout distort1 gadist, kpre, kpost, kshap1, kshap2, 1

  outs aout, aout

  gadist = 0
endin

</CsInstruments>
<CsScore>

; Sta Dur Amp Pitch
i1 0.0 3.0 0.5 6.00
i1 0.5 2.5 0.5 7.00
i1 1.0 2.0 0.5 7.07
i1 1.5 1.5 0.5 8.00

; Sta Dur PreGain PostGain Shape1 Shape2
i50 0 4 2 .5 0 0
e

</CsScore>
</CsoundSynthesizer>

```

## Credits

Author: Hans Mikelson  
December 1998

New in Csound version 3.50

# divz

divz — Safely divides two numbers.

## Syntax

```
ares divz xa, xb, ksubst
ires divz ia, ib, isubst
kres divz ka, kb, ksubst
...divz(ka, kb, ksubst)... (no rate restriction)
```

## Description

Safely divides two numbers.

## Initialization

Whenever  $b$  is not zero, set the result to the value  $a/b$ ; when  $b$  is zero, set it to the value of *subst* instead.

## Examples

Here is an example of the divz opcode. It uses the file *divz.csd* [examples/divz.csd].

### Example 218. Example of the divz opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o divz.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Define the numbers to be divided.
ka init 200
; Linearly change the value of kb from 200 to 0.
kb line 0, p3, 200
; If a "divide by zero" error occurs, substitute -1.
ksubst init -1

; Safely divide the numbers.
```

```
kresults divz ka, kb, ksubst  
  
; Print out the results.  
printks "%f / %f = %f\\n", 0.1, ka, kb, kresults  
endin  
  
</CsInstruments>  
<CsScore>  
  
; Play Instrument #1 for one second.  
i 1 0 1  
e  
  
</CsScore>  
</CsoundSynthesizer>
```

Its output should include lines like:

```
200.000000 / 0.000000 = -1.000000  
200.000000 / 19.999887 = 10.000056  
200.000000 / 40.000027 = 4.999997
```

## See Also

*=, init, tival*

## Credits

Author: John ffitich after an idea by Barry L. Vercoe

Example written by Kevin Conder.

Functional form available in csound Version 6.00

# doppler

`doppler` — A fast and robust method for approximating sound propagation, achieving convincing Doppler shifts without having to solve equations.

## Description

A fast and robust method for approximating sound propagation, achieving convincing Doppler shifts without having to solve equations. The method computes frequency shifts based on reading an input delay line at a delay time computed from the distance between source and mic and the speed of sound. One instance of the opcode is required for each dimension of space through which the sound source moves. If the source sound moves at a constant speed from in front of the microphone, through the microphone, to behind the microphone, then the output will be frequency shifted above the source frequency at a constant frequency while the source approaches, then discontinuously will be shifted below the source frequency at a constant frequency as the source recedes from the microphone. If the source sound moves at a constant speed through a point to one side of the microphone, then the rate of change of position will not be constant, and the familiar Doppler frequency shift typical of a siren or engine approaching and receding along a road beside a listener will be heard.

## Syntax

```
ashifted doppler asource, ksourceposition, kmicposition [, isoundspeed, ifiltercutoff]
```

## Initialization

*isoundspeed* (optional, default=340.29) -- Speed of sound in meters/second.

*ifiltercutoff* (optional, default=6) -- Rate of updating the position smoothing filter, in cycles/second.

## Performance

*asource* -- Input signal at the sound source.

*ksourceposition* -- Position of the source sound in meters. The distance between source and mic should not be changed faster than about 3/4 the speed of sound.

*kmicposition* -- Position of the recording microphone in meters. The distance between source and mic should not be changed faster than about 3/4 the speed of sound.

## Examples

Here is an example of the `doppler` opcode. It uses the file `doppler.csd` [examples/doppler.csd].

### Example 219. Example of the doppler opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform
```

```

-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o doppler.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 48000
ksmps = 128
nchnls = 2
Odbfs = 1

instr 1

iattack    init    0.05
irelease   init    0.05
isustain   init    p3
p3         init    iattack + isustain + irelease
kdamping   linseg  0.0, iattack, 1.0, isustain, 1.0, irelease, 0.0
kmic       init    4
           ; Position envelope, with a changing rate of change of position.
           transeg a dur ty b dur ty c dur ty d
kposition  transeg 4, p3*.4, 0, 120, p3*.3, -3, 50, p3*.3, 2, 4
ismoothinghz init 6
ispeedofsound init 340.29
asignal    vco2    0.5, 110
aoutput    doppler asignal, kposition, kmic, ispeedofsound, ismoothinghz
           outs    aoutput*kdamping, aoutput * kdamping

endin

</CsInstruments>
<CsScore>

i1 0.0 20
e1
</CsScore>
</CsoundSynthesizer>

```

## Credits

Author of algorithm: Peter Brinkmann  
 Author of opcode: Michael Gogins  
 January 2010

New in Csound version 5.11

# dot

dot — Calculates the dot product of two arrays.

## Description

Takes two numeric arrays (k or i-rate) and calculates the dot product.

## Syntax

```
kres/iresdot karr1[]/iarr1[], karr2[]/iarr2[] (k- or i-arrays )
```

## Examples

Here is an example of the dot opcode. It uses the file *dota.csd* [examples/dot.csd].

### Example 220. Example of the dot opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-n
</CsOptions>
<CsInstruments>

instr 1
  kArr1[] fillarray 1,3,2,7,4
  kArr2[] fillarray 2,3,1,2,5
  kd dot kArr1,kArr2
  printk2 kd
  turnoff
endin

</CsInstruments>
<CsScore>
i1 0 1
</CsScore>
</CsoundSynthesizer>
```

## See Also

*sumarray product*

## Credits

Author: Victor Lazzarini

New in version 6.09

# downsamp

downsamp — Modify a signal by down-sampling.

## Description

Modify a signal by down-sampling.

## Syntax

```
kres downsamp asig [, iwlen]
```

## Initialization

*iwlen* (optional) -- window length in samples over which the audio signal is averaged to determine a downsampled value. Maximum length is *ksmps*; 0 and 1 imply no window averaging. The default value is 0.

## Performance

*downsamp* converts an audio signal to a control signal by downsampling. It produces one kval for each audio control period. The optional window invokes a simple averaging process to suppress foldover.

## Examples

Here is an example of the *downsamp* opcode. It uses the file *downsamp.csd* [examples/downsamp.csd].

### Example 221. Example of the *downsamp* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o downsamp.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifrq = cpspch(p4)
ain diskin2 "beats.wav", 1
aenv follow ain, .001 ;take the amplitude every 1/1000th of a second
alow tone aenv, 25 ;lowpass-filter (25 Hz) for a clean signal
kenv downsamp allow
```

```
asig   pluck   kenv, ifrq, 15, 0, 1
outs  asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 2 9
i 1 + . 7
i 1 + . 5

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*diff, integ, interp, samphold, upsamp*

# dripwater

dripwater — Semi-physical model of a water drop.

## Description

*dripwater* is a semi-physical model of a water drop. It is one of the PhISEM percussion opcodes. PhISEM (Physically Informed Stochastic Event Modeling) is an algorithmic approach for simulating collisions of multiple independent sound producing objects.

## Syntax

```
ares dripwater kamp, idettack [, inum] [, idamp] [, imaxshake] [, ifreq] \  
    [, ifreq1] [, ifreq2]
```

## Initialization

*idettack* -- period of time over which all sound is stopped

*inum* (optional) -- The number of beads, teeth, bells, timbrels, etc. If zero, the default value is 10.

*idamp* (optional) -- the damping factor, as part of this equation:

$$\text{damping\_amount} = 0.996 + (\text{idamp} * 0.002)$$

The default *damping\_amount* is 0.996 which means that the default value of *idamp* is 0. The maximum *damping\_amount* is 1.0 (no damping). This means the maximum value for *idamp* is 2.0.

The recommended range for *idamp* is usually below 75% of the maximum value. Rasmus Ekman suggests a range of 1.4-1.75. He also suggests a maximum value of 1.9 instead of the theoretical limit of 2.0.

*imaxshake* (optional, default=0) -- amount of energy to add back into the system. The value should be in range 0 to 1.

*ifreq* (optional) -- the main resonant frequency. The default value is 450.

*ifreq1* (optional) -- the first resonant frequency. The default value is 600.

*ifreq2* (optional) -- the second resonant frequency. The default value is 750.

## Performance

*kamp* -- Amplitude of output. Note: As these instruments are stochastic, this is only an approximation.

## Examples

Here is an example of the dripwater opcode. It uses the file *dripwater.csd* [examples/dripwater.csd].

### Example 222. Example of the dripwater opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o dripwater.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

imaxshake = p4
ifreq      = p5
ifreq1     = p6
ifreq2     = p7

;low amplitude
adrp dripwater .1, 0.09, 10, .9, imaxshake, ifreq, ifreq1, ifreq2
asig clip adrp, 2, 0.9 ; avoid drips that drip too loud
outs asig, asig

endin
</CsInstruments>
<CsScore>

{100 CNT

i1 [0.1 * $CNT] 0.5 0.5 430 1000 800

}

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*bamboo, guiro, sleighbells, tambourine*

## Credits

Author: Perry Cook, part of the PhISEM (Physically Informed Stochastic Event Modeling)

Adapted by John ffitch

University of Bath, Codemist Ltd.

Bath, UK

New in Csound version 4.07

Added notes by Rasmus Ekman on May 2002.

# dssiactivate

`dssiactivate` — Activates or deactivates a DSSI or LADSPA plugin.



## Note

plugin: Requires `dssi4cs` plugin

## Syntax

```
dssiactivate ihandle, ktoggle
```

## Description

`dssiactivate` is used to activate or deactivate a DSSI or LADSPA plugin. It calls the plugin's `activate()` and `deactivate()` functions if they are provided.

## Initialization

*ihandle* - the number which identifies the plugin, generated by `dssiinit`.

## Performance

*ktoggle* - Selects between activation (`ktoggle=1`) and deactivation (`ktoggle=0`).

`dssiactivate` is used to turn on and off plugins if they provide this facility. This may help conserve CPU processing in some cases. For consistency, all plugins must be activated to produce sound. An inactive plugin produces silence.

Depending on the plugin's implementation, this may cause interruptions in the realtime audio process, so use with caution.

`dssiactivate` may cause audio stream breakups when used in realtime, so it is recommended to load all plugins to be used before playing.



## Warning

Please note that even if `activate()` and `deactivate()` functions are not present in a plugin, `dssiactivate` must be called for the plugin to produce sound.

## Examples

Here is an example of the `dssiactivate` opcode. It uses the file `dssiactivate.csd` [examples/dssiactivate.csd].

### Example 223. Example of the `dssiactivate` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
```

```

<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o dssiactivate.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

gihandle dssiinit "caps.so", 19, 1 ; = mono phaser and
gaout init 0 ; verbose about all ports

instr 1 ; activate DSSI

ktoggle = p4
dssiactivate gihandle, ktoggle
endin

instr 2

ain1 diskin2 "beats.wav", 1,0,1 ; loop
ain1 = ain1*.5
outs ain1, ain1
gaout = gaout+ain1
endin

instr 3

dssictls gihandle, 0, 1, 1 ; range -1 to 1
dssictls gihandle, 1, 2, 1 ; rate 0 to 10
dssictls gihandle, 2, .8, 1 ; depth 0 to 1
dssictls gihandle, 3, 3, 1 ; spread 0 to 3.14
dssictls gihandle, 4, .9, 1 ; feedback 0 to 0.999

endin

instr 4

aout1 dssiaudio gihandle, gaout ;get beats.wav, mono out
outs aout1,aout1

gaout = 0
endin

</CsInstruments>
<CsScore>
i 1 0 4 1
i 1 + . 0
i 1 + . 1
i 1 + . 0
i 1 + . 1
i 2 1 20
i 3 1 20
i 4 0 20

e
</CsScore>
</CsSoundSynthesizer>

```

## Credits

2005

By: Andrés Cabrera

Uses code from Richard Furse's LADSPA sdk.

# dssiaudio

dssiaudio — Processes audio using a LADSPA or DSSI plugin.



## Note

plugin: Requires dssi4cs plugin

## Syntax

```
[aout1, aout2, ..., aout9] dssiaudio ihandle, [ain1, ain2, ..., ain9]
```

## Description

*dssiaudio* generates audio by processing an input signal through a LADSPA plugin.

## Initialization

*ihandle* - handle for the plugin returned by *dssiinit*

## Performance

*aout1, aout2, etc* - Audio output generated by the plugin

*ain1, ain2, etc* - Audio provided to the plugin for processing

*dssiaudio* runs a plugin on the provided audio and produces audio output. Currently upto four inputs and outputs are provided. You should provide signal for all the plugins audio inputs, otherwise unpredictable results may occur. If the plugin doesn't have any input (e.g Noise generator) you must still provide at least one input variable, which will be ignored with a message.

Only one *dssiaudio* should be executed once per plugin, or strange results may occur.

## Examples

Here is an example of the dssiaudio opcode. It uses the file *dssiaudio.csd* [examples/dssiaudio.csd].

### Example 224. Example of the dssiaudio opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o DSSIplay_mono.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```
sr = 44100
ksmps = 32
nchnls = 2

gihandle dssiinit "caps.so", 19, 1 ; = mono phaser and
gaout  init 0 ; verbose about all ports

instr 1 ; activate DSSI

dssiactivate gihandle, 1
endin

instr 2
ain1 diskin2 "beats.wav", 1,0,1 ; loop

gaout = gaout+(ain1*.5)
endin

instr 3

dssictls gihandle, 0, .8, 1 ; range -1 to 1
dssictls gihandle, 1, .05, 1 ; rate 0 to 10
dssictls gihandle, 2, .8, 1 ; depth 0 to 1
dssictls gihandle, 3, 2, 1 ; spread 0 to 3.14
dssictls gihandle, 4, .7, 1 ; feedback 0 to 0.999

endin

instr 4

aout1 dssiaudio gihandle, gaout ;get beats.wav, mono out
outs aout1,aout1

gaout = 0

endin
</CsInstruments>
<CsScore>
i 1 0 20
i 2 1 20
i 3 1 20
i 4 0 20

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

2005

By: Andrés Cabrera

Uses code from Richard Furse's LADSPA sdk.

# dssictls

dssictls — Send control information to a LADSPA or DSSI plugin.



## Note

plugin: Requires dssi4cs plugin

## Syntax

```
dssictls ihandle, iport, kvalue, ktrigger
```

## Description

*dssictls* sends control values to a plugin's control port

## Initialization

*ihandle* - handle for the plugin returned by *dssiinit*

*iport* - control port number

## Performance

*kvalue* - value to be assigned to the port

*ktrigger* - determines whether the control information will be sent (*ktrigger* = 1) or not. This is useful for thinning control information, generating *ktrigger* with *metro*

*dssictls* sends control information to a LADSPA or DSSI plugin's control port. The valid control ports and ranges are given by *dssiinit*. Using values outside the ranges may produce unspecified behaviour.

## Examples

Here is an example of the *dssictls* opcode. It uses the file *dssictls.csd* [examples/dssictls.csd].

### Example 225. Example of the *dssictls* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o dssictls.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
```

```

ksmps = 32
nchnls = 2

gihandle dssiinit "caps.so", 1, 1 ; = equaliser and
gaoutl init 0 ; verbose about all ports
gaoutr init 0

instr 1 ; activate DSSI

dssiactivate gihandle, 1
endin

instr 2
ainl diskin2 "beats.wav", 1,0,1 ; loop

gaoutl = gaoutl+(ainl*.1) ; temper input
gaoutr = gaoutr+(ainl*.1)
endin

instr 3

dssictls gihandle, 2, -48, 1 ; 31 Hz range -48 to 24
dssictls gihandle, 3, -48, 1 ; 63 Hz range -48 to 24
dssictls gihandle, 4, -48, 1 ; 125 Hz range -48 to 24
dssictls gihandle, 5, 20, 1 ; 250 Hz range -48 to 24
dssictls gihandle, 6, -48, 1 ; 500 Hz range -48 to 24
dssictls gihandle, 7, -48, 1 ; 1 kHz Hz range -48 to 24
dssictls gihandle, 8, -48, 1 ; 2 kHz range -48 to 24
dssictls gihandle, 9, 24, 1 ; 4 kHz range -48 to 24
dssictls gihandle, 10, 24, 1 ; 8 kHz range -48 to 24
dssictls gihandle, 11, 24, 1 ; 16 kHz range -48 to 24

endin

instr 4

aout1, aout2 dssiaudio gihandle, gaoutl, gaoutr ;get beats.wav, mono out
outs aout1,aout2

gaoutl = 0
gaoutr = 0
endin

</CsInstruments>
<CsScore>
i 1 0 20
i 2 1 20
i 3 1 20
i 4 0 20

e
</CsScore>
</CsoundSynthesizer>

```

## Credits

2005

By: Andrés Cabrera

Uses code from Richard Furse's LADSPA sdk.

# dssiinit

dssiinit — Loads a DSSI or LADSPA plugin.



## Note

plugin: Requires dssi4cs plugin

## Syntax

```
ihandle dssiinit ilibraryname, ipluginindex [, iverbose]
```

## Description

*dssiinit* is used to load a DSSI or LADSPA plugin into memory for use with the other dssi4cs opcodes. Both LADSPA effects and DSSI instruments can be used.

## Initialization

*ihandle* - the number which identifies the plugin, to be passed to other dssi4cs opcodes.

*ilibraryname* - the name of the .so (shared object) file to load.

*ipluginindex* - The index of the plugin to be used.

*iverbose* (optional) - show plugin information and parameters when loading. (default = 1)

*dssiinit* looks for *ilibraryname* on LADSPA\_PATH and DSSI\_PATH. One of these variables must be set, otherwise *dssiinit* will return an error. LADSPA and DSSI libraries may contain more than one plugin which must be referenced by its index. *dssiinit* then attempts to find plugin index *ipluginindex* in the library and load the plugin into memory if it is found. To find out which plugins you have available and their index numbers you can use: *dssilist*.

If *iverbose* is not 0 (the default), information about the plugin detailing its characteristics and its ports will be shown. This information is important for opcodes like *dssiactls*.

Plugins are set to inactive by default, so you *must* use *dssiactivate* to get the plugin to produce sound. This is required even if the plugin doesn't provide an activate() function.

*dssiinit* may cause audio stream breakups when used in realtime, so it is recommended to load all plugins to be used before playing.

## Examples

Here is an example of the dssinit opcode. It uses the file *dssiinit.csd* [examples/dssiinit.csd].

**Example 226. Example of the dssiinit opcode. (Remember to change the Library name)**

```
<CsoundSynthesizer>
```

```

<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o dssiinit.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

gihandle dssiinit "g2reverb.so", 0, 1
gaout    init 0

instr 1 ; activate DSSI

dssiactivate gihandle, 1
endin

instr 2

ain1 diskin2 "beats.wav", 1

gaout = gaout+(ain1*.3)
endin

instr 3

dssictls gihandle, 4, 100, 1 ; room 10 to 150
dssictls gihandle, 5, 10, 1 ; reverb time 1 to 20
dssictls gihandle, 6, .5, 1 ; input bandwidth 0 to 1
dssictls gihandle, 7, .25, 1 ; damping 0 to 1
dssictls gihandle, 8, 0, 1 ; dry -80 to 0
dssictls gihandle, 9, -10, 1 ; reflections -80 to 0
dssictls gihandle, 10, -15, 1 ; rev. tail -80 to 0
endin

instr 4

aout1, aout2 dssiaudio gihandle, gaout, gaout ;get beats.wav and
             outs aout1,aout2 ; stereo DSSI plugin

gaout = 0
endin
</CsInstruments>
<CsScore>
i 1 0 2
i 2 1 10
i 3 1 10
i 4 0 10
e
</CsScore>
</CsoundSynthesizer>

```

## Credits

2005

By: Andrés Cabrera

Uses code from Richard Furse's LADSPA sdk.

# dssilist

dssilist — Lists all available DSSI and LADSPA plugins.



## Note

plugin: Requires dssi4cs plugin

## Syntax

```
dssilist
```

## Description

*dssilist* checks the variables DSSI\_PATH and LADSPA\_PATH and lists all plugins available in all plugin libraries there.

LADSPA and DSSI libraries may contain more than one plugin which must be referenced by the index provided by *dssilist*.

This opcode produces a long printout which may interrupt realtime audio output, so it should be run at the start of a performance.

## Examples

Here is an example of the dssilist opcode. It uses the file *dssilist.csd* [examples/dssilist.csd].

### Example 227. Example of the dssilist opcode. (Remember to change the Library name)

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out
-odac

</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

instr 1 ; list all DSSI and LADSPA plugins

dssilist

endin
</CsInstruments>
<CsScore>
i 1 0 0

e
</CsScore>
```

</CsoundSynthesizer>

## Credits

2005

By: Andrés Cabrera

Uses code from Richard Furse's LADSPA sdk.

# dumpk

dumpk — Periodically writes an orchestra control-signal value to an external file.

## Description

Periodically writes an orchestra control-signal value to a named external file in a specific format.

## Syntax

```
dumpk ksig, ifilename, iformat, iprd
```

## Initialization

*ifilename* -- character string (in double quotes, spaces permitted) denoting the external file name. May either be a full path name with target directory specified or a simple filename to be created within the current directory

*iformat* -- specifies the output data format:

- 1 = 8-bit signed char(high order 8 bits of a 16-bit integer)
- 4 = 16-bit short integers
- 5 = 32-bit long integers
- 6 = 32-bit floats
- 7 = ASCII long integers
- 8 = ASCII floats (2 decimal places)

Note that A-law and U-law output are not available, and that all formats except the last two are binary. The output file contains no header information.

*iprd* -- the period of *ksig* output in seconds, rounded to the nearest orchestra control period. A value of 0 implies one control period (the enforced minimum), which will create an output file sampled at the orchestra control rate.

## Performance

*ksig* -- a control-rate signal

This opcode allows a generated control signal value to be saved in a named external file. The file contains no self-defining header information. But it contains a regularly sampled time series, suitable for later input or analysis. There may be any number of *dumpk* opcodes in an instrument or orchestra but each must write to a different file.

## Examples

Here is an example of the *dumpk* opcode. It uses the file *dumpk.csd* [examples/dumpk.csd]. Note that the example must be run from a directory for which the user has write access.

**Example 228. Example of the dumpk opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o dumpk.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 20
nchnls = 1

; By Andres Cabrera 2008

instr 1
; Write fibonacci numbers to file "fibonacci.txt"
; as ascii long integers (mode 7), using the orchestra's
; control rate (iprd = 0)

knumber init 0
koldnumber init 1
ktrans init 1
ktrans = knumber
knumber = knumber + koldnumber
koldnumber = ktrans
dumpk knumber, "fibonacci.txt", 7, 0
printk2 knumber
endin

</CsInstruments>
<CsScore>

;Write to the file for 1 second. Since control rate is 20, 20 values will be written
i 1 0 1

</CsScore>
</CsoundSynthesizer>
```

Here is another example of the dumpk opcode. It uses the file *dumpk-2.csd* [examples/dumpk-2.csd].

**Example 229. Example 2 of the dumpk opcode.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc      ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o dumpk-2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine ftgen 0, 0, 2^10, 10, 1

instr 1 ;writes a control signal to a file
kfreq randh 100, 1, 2, 1, 500 ;generates one random number between 400 and 600 per second
dumpk kfreq, "dumpk.txt", 8, 1 ;writes the control signal
printk 1, kfreq ;prints it
endin

instr 2 ;reads the file written by instr 1
kfreq readk "dumpk.txt", 8, 1
printk 1, kfreq ;prints it
aout poscil .2, kfreq, giSine
outs aout, aout
endin

</CsInstruments>
<CsScore>
i 1 0 5
i 2 5 5
e
</CsScore>
</CsSoundSynthesizer>
```

Its output should include lines like this:

```
WARNING: Seeding from current time 683384022
```

```
i 1 time 1.00033: 463.64510
i 1 time 2.00000: 463.64510
i 1 time 3.00000: 483.14200
i 1 time 4.00000: 567.55973
i 1 time 5.00000: 576.37060
i 1 time 6.00000: 460.66550

i 2 time 6.00033: 463.64510
i 2 time 7.00000: 463.64510
i 2 time 8.00000: 483.14200
i 2 time 9.00000: 567.55970
i 2 time 10.00000: 576.37060
i 2 time 11.00000: 460.66550
```

## See Also

*dumpk2, dumpk3, dumpk4, readk, readk2, readk3, readk4*

## Credits

By: John ffitich and Barry L. Vercoe

1999 or earlier

# dumpk2

dumpk2 — Periodically writes two orchestra control-signal values to an external file.

## Description

Periodically writes two orchestra control-signal values to a named external file in a specific format.

## Syntax

```
dumpk2 ksig1, ksig2, ifilename, iformat, iprd
```

## Initialization

*ifilename* -- character string (in double quotes, spaces permitted) denoting the external file name. May either be a full path name with target directory specified or a simple filename to be created within the current directory

*iformat* -- specifies the output data format:

- 1 = 8-bit signed char(high order 8 bits of a 16-bit integer)
- 4 = 16-bit short integers
- 5 = 32-bit long integers
- 6 = 32-bit floats
- 7 = ASCII long integers
- 8 = ASCII floats (2 decimal places)

Note that A-law and U-law output are not available, and that all formats except the last two are binary. The output file contains no header information.

*iprd* -- the period of *ksig* output in seconds, rounded to the nearest orchestra control period. A value of 0 implies one control period (the enforced minimum), which will create an output file sampled at the orchestra control rate.

## Performance

*ksig1*, *ksig2* -- control-rate signals.

This opcode allows two generated control signal values to be saved in a named external file. The file contains no self-defining header information. But it contains a regularly sampled time series, suitable for later input or analysis. There may be any number of *dumpk2* opcodes in an instrument or orchestra but each must write to a different file.

## Examples

Here is an example of the *dumpk2* opcode. It uses the file *dumpk2.csd* [examples/dumpk2.csd]. Note that the example must be run from a directory for which the user has write access.

**Example 230. Example of the dumpk2 opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;;realtime audio out
;-iadc     ;;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o dumpk2.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine ftgen 0, 0, 2^10, 10, 1

instr 1 ;writes two control signals to a file
kfreq  randh 100, 1, 2, 1, 500 ;generates one random number between 400 and 600 per second
kdb    randh 12, 1, 2, 1, -12 ;amplitudes in dB between -24 and 0
      dumpk2 kfreq, kdb, "dumpk2.txt", 8, 1 ;writes the control signals
      prints "WRITING:\n"
      printks "kfreq = %f, kdb = %f\n", 1, kfreq, kdb ;prints them
endin

instr 2 ;reads the file written by instr 1
kf,kdb readk2 "dumpk2.txt", 8, 1
      prints "READING:\n"
      printks "kfreq = %f, kdb = %f\n", 1, kf, kdb ;prints again
kdb    lineto kdb, .1 ;smoothing amp transition
aout   poscil ampdb(kdb), kf, giSine
      outs  aout, aout
endin

</CsInstruments>
<CsScore>
i 1 0 5
i 2 5 5
e
</CsScore>
</CsoundSynthesizer>

```

The output should include lines like these:

```

kfreq = 429.202551, kdb = -20.495694
kfreq = 429.202551, kdb = -20.495694
kfreq = 407.275258, kdb = -23.123776
kfreq = 475.264472, kdb = -9.300846
kfreq = 569.979181, kdb = -7.315527
kfreq = 440.103457, kdb = -0.058331

kfreq = 429.202600, kdb = -20.495700
kfreq = 429.202600, kdb = -20.495700
kfreq = 407.275300, kdb = -23.123800
kfreq = 475.264500, kdb = -9.300800
kfreq = 569.979200, kdb = -7.315500
kfreq = 440.103500, kdb = -0.058300

```

## See Also

*dumpk, dumpk3, dumpk4, readk, readk2, readk3, readk4*

## Credits

By: John ffitich and Barry L. Vercoe

1999 or earlier

# dumpk3

dumpk3 — Periodically writes three orchestra control-signal values to an external file.

## Description

Periodically writes three orchestra control-signal values to a named external file in a specific format.

## Syntax

```
dumpk3 ksig1, ksig2, ksig3, ifilename, iformat, iprd
```

## Initialization

*ifilename* -- character string (in double quotes, spaces permitted) denoting the external file name. May either be a full path name with target directory specified or a simple filename to be created within the current directory

*iformat* -- specifies the output data format:

- 1 = 8-bit signed char(high order 8 bits of a 16-bit integer)
- 4 = 16-bit short integers
- 5 = 32-bit long integers
- 6 = 32-bit floats
- 7 = ASCII long integers
- 8 = ASCII floats (2 decimal places)

Note that A-law and U-law output are not available, and that all formats except the last two are binary. The output file contains no header information.

*iprd* -- the period of *ksig* output in seconds, rounded to the nearest orchestra control period. A value of 0 implies one control period (the enforced minimum), which will create an output file sampled at the orchestra control rate.

## Performance

*ksig1*, *ksig2*, *ksig3* -- control-rate signals

This opcode allows three generated control signal values to be saved in a named external file. The file contains no self-defining header information. But it contains a regularly sampled time series, suitable for later input or analysis. There may be any number of *dumpk3* opcodes in an instrument or orchestra but each must write to a different file.

## Examples

Here is an example of the *dumpk3* opcode. It uses the file *dumpk3.csd* [examples/dumpk3.csd]. Note that the example must be run from a directory for which the user has write access.

**Example 231. Example of the dumpk3 opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o dumpk3.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine ftgen 0, 0, 2^10, 10, 1

instr 1 ;writes three control signals to a file
kfreq   randh   100, 1, 2, 1, 500 ;generates one random number between 400 and 600 per second
kdb     randh   12, 1, 2, 1, -12 ;amplitudes in dB between -24 and 0
kpan    randh   .5, 1, 2, 1, .5 ;panning between 0 and 1
        dumpk3  kfreq, kdb, kpan, "dumpk3.txt", 8, 1 ;writes the control signals
        prints  "WRITING:\n"
        printks "kfreq = %f, kdb = %f, kpan = %f\n", 1, kfreq, kdb, kpan ;prints them
endin

instr 2 ;reads the file written by instr 1
kf,kdb,kp readk3 "dumpk3.txt", 8, 1
        prints  "READING:\n"
        printks "kfreq = %f, kdb = %f, kpan = %f\n", 1, kf, kdb, kp ;prints again
kdb     lineto  kdb, .1 ;smoothing amp transition
kp      lineto  kp, .1 ;smoothing pan transition
aout    poscil  ampdb(kdb), kf, giSine
aL, aR  pan2    aout, kp
        outs   aL, aR
endin

</CsInstruments>
<CsScore>
i 1 0 5
i 2 5 5
e
</CsScore>
</CsoundSynthesizer>

```

The output should include lines like these:

```

WRITING:
kfreq = 473.352855, kdb = -15.197657, kpan = 0.366764
kfreq = 473.352855, kdb = -15.197657, kpan = 0.366764
kfreq = 441.426368, kdb = -19.026206, kpan = 0.207327
kfreq = 452.965140, kdb = -21.447486, kpan = 0.553270
kfreq = 585.106328, kdb = -11.903852, kpan = 0.815665
kfreq = 482.056760, kdb = -4.046744, kpan = 0.876537

READING:
kfreq = 473.352900, kdb = -15.197700, kpan = 0.366800
kfreq = 473.352900, kdb = -15.197700, kpan = 0.366800
kfreq = 441.426400, kdb = -19.026200, kpan = 0.207300

```

kfreq = 452.965100, kdb = -21.447500, kpan = 0.553300  
kfreq = 585.106300, kdb = -11.903900, kpan = 0.815700  
kfreq = 482.056800, kdb = -4.046700, kpan = 0.876500

## See Also

*dumpk, dumpk2, dumpk4, readk, readk2, readk3, readk4*

## Credits

By: John ffitich and Barry L. Vercoe

1999 or earlier

# dumpk4

dumpk4 — Periodically writes four orchestra control-signal values to an external file.

## Description

Periodically writes four orchestra control-signal values to a named external file in a specific format.

## Syntax

```
dumpk4 ksig1, ksig2, ksig3, ksig4, ifilename, iformat, iprd
```

## Initialization

*ifilename* -- character string (in double quotes, spaces permitted) denoting the external file name. May either be a full path name with target directory specified or a simple filename to be created within the current directory

*iformat* -- specifies the output data format:

- 1 = 8-bit signed char(high order 8 bits of a 16-bit integer)
- 4 = 16-bit short integers
- 5 = 32-bit long integers
- 6 = 32-bit floats
- 7 = ASCII long integers
- 8 = ASCII floats (2 decimal places)

Note that A-law and U-law output are not available, and that all formats except the last two are binary. The output file contains no header information.

*iprd* -- the period of *ksig* output in seconds, rounded to the nearest orchestra control period. A value of 0 implies one control period (the enforced minimum), which will create an output file sampled at the orchestra control rate.

## Performance

*ksig1, ksig2, ksig3, ksig4* -- control-rate signals

This opcode allows four generated control signal values to be saved in a named external file. The file contains no self-defining header information. But it contains a regularly sampled time series, suitable for later input or analysis. There may be any number of *dumpk4* opcodes in an instrument or orchestra but each must write to a different file.

## Examples

Here is an example of the *dumpk4* opcode. It uses the file *dumpk4.csd* [examples/dumpk4.csd]. Note that the example must be run from a directory for which the user has write access.

**Example 232. Example of the dumpk4 opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o dumpk4.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine ftgen 0, 0, 2^10, 10, 1

instr 1 ;writes four control signals to a file
kcf      randh      950, 1, 2, 1, 1050 ;generates one random number between 100 and 2000 per second
kq       randh      10, 1, 2, 1, 11 ;generates another random number between 1 and 21 per second
kdb      randh      9, 1, 2, 1, -15 ;amplitudes in dB between -24 and -6
kpan     randh      .5, 1, 2, 1, .5 ;panning between 0 and 1
         dumpk4     kcf, kq, kdb, kpan, "dumpk4.txt", 8, 1 ;writes the control signals
         prints     "WRITING:\n"
         printks   "kcf = %f, kq = %f, kdb = %f, kpan = %f\n", 1, kcf, kq, kdb, kpan ;prints them
endin

instr 2 ;reads the file written by instr 1
kcf,kq,kdb,kp readk4 "dumpk4.txt", 8, 1
         prints     "READING:\n"
         printks   "kcf = %f, kq = %f, kdb = %f, kpan = %f\n", 1, kcf, kq, kdb, kp ;prints values
kdb      lineto     kdb, .1 ;smoothing amp transition
kp       lineto     kp, .1 ;smoothing pan transition
anoise   rand       ampdb(kdb), 2, 1
kbw      =          kcf/kq ;bandwidth of resonant filter
abp      reson     anoise, kcf, kbw
aout     balance    abp, anoise
aL, aR   pan2      aout, kp
         outs      aL, aR
endin

</CsInstruments>
<CsScore>
i 1 0 5
i 2 5 5
e
</CsScore>
</CsoundSynthesizer>

```

The output should include lines like these:

```

WRITING:
kcf = 1122.469723, kq = 11.762839, kdb = -14.313445, kpan = 0.538142
kcf = 1122.469723, kq = 11.762839, kdb = -14.313445, kpan = 0.538142
kcf = 1148.638412, kq = 12.040490, kdb = -14.061868, kpan = 0.552205
kcf = 165.796855, kq = 18.523179, kdb = -15.816977, kpan = 0.901528
kcf = 147.729960, kq = 13.071911, kdb = -11.924531, kpan = 0.982518
kcf = 497.430113, kq = 13.605512, kdb = -21.586611, kpan = 0.179229

```

READING:

WARNING: Seeding from current time 3308160476

kcf = 1122.469700, kq = 11.762800, kdb = -14.313400, kpan = 0.538100

kcf = 1122.469700, kq = 11.762800, kdb = -14.313400, kpan = 0.538100

kcf = 1148.638400, kq = 12.040500, kdb = -14.061900, kpan = 0.552200

kcf = 165.796900, kq = 18.523200, kdb = -15.817000, kpan = 0.901500

kcf = 147.730000, kq = 13.071900, kdb = -11.924500, kpan = 0.982500

kcf = 497.430100, kq = 13.605500, kdb = -21.586600, kpan = 0.179200

## See Also

*dumpk, dumpk2, dumpk3, readk, readk2, readk3, readk4*

## Credits

By: John ffitich and Barry L. Vercoe

1999 or earlier

# duserrnd

duserrnd — Discrete USER-defined-distribution RaNDom generator.

## Description

Discrete USER-defined-distribution RaNDom generator.

## Syntax

```
aout duserrnd ktableNum
```

```
iout duserrnd itableNum
```

```
kout duserrnd ktableNum
```

## Initialization

*itableNum* -- number of table containing the random-distribution function. Such table is generated by the user. See GEN40, GEN41, and GEN42. The table length does not need to be a power of 2

## Performance

*ktableNum* -- number of table containing the random-distribution function. Such table is generated by the user. See GEN40, GEN41, and GEN42. The table length does not need to be a power of 2

*duserrnd* (discrete user-defined-distribution random generator) generates random values according to a discrete random distribution created by the user. The user can create the discrete distribution histogram by using GEN41. In order to create that table, the user has to define an arbitrary amount of number pairs, the first number of each pair representing a value and the second representing its probability (see GEN41 for more details).

When used as a function, the rate of generation depends by the rate type of input variable XtableNum. In this case it can be embedded into any formula. Table number can be varied at k-rate, allowing to change the distribution histogram during the performance of a single note. *duserrnd* is designed be used in algorithmic music generation.

*duserrnd* can also be used to generate values following a set of ranges of probabilities by using distribution functions generated by GEN42 (See GEN42 for more details). In this case, in order to simulate continuous ranges, the length of table XtableNum should be reasonably big, as *duserrnd* does not interpolate between table elements.

For a tutorial about random distribution histograms and functions see:

- D. Lorrain. "A panoply of stochastic cannons". In C. Roads, ed. 1989. Music machine. Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the duserrnd opcode. It uses the file *duserrnd.csd* [examples/duserrnd.csd].

### Example 233. Example of the `duserrnd` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o duserrnd.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

k1  duserrnd 1
    printk 0, k1
asig poscil .5, 220*k1, 2 ;multiply frequency with random value
outs asig, asig

endin
</CsInstruments>
<CsScore>
f1 0 -20 -41 2 .1 8 .9 ;choose 2 at 10% probability, and 8 at 90%

f2 0 8192 10 1

i1 0 2
e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like these:

```
i 1 time 0.00067: 8.00000
i 1 time 0.00133: 8.00000
i 1 time 0.00200: 8.00000
i 1 time 0.00267: 8.00000
i 1 time 0.00333: 2.00000
i 1 time 0.00400: 8.00000
i 1 time 0.00533: 8.00000
i 1 time 0.00600: 8.00000
.....
```

## See Also

*cuserrnd*, *urd*

## Credits

Author: Gabriel Maldonado

New in Version 4.16

# dust

dust — Random impulses.

## Description

Generates random impulses from 0 to 1.

## Syntax

```
ares dust kamp, kdensity
kres dust kamp, kdensity
```

## Performance

*kamp* -- amplitude.

*kdensity* -- average number of impulses per second.

## Examples

Here is an example of the dust opcode. It uses the file *dust.csd* [examples/dust.csd].

### Example 234. Example of the dust opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o oscil.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kdens expon 2, p3, 20000
aout dust 0.5, kdens
outs aout, aout

endin
</CsInstruments>
<CsScore>
i1 0 10
e
</CsScore>
```

`</CsoundSynthesizer>`

## See Also

*dust2 gausstrig mpulse*

## Credits

Based on James McCartney's Dust ugen (SuperCollider)

Author: Tito Latini

January 2012

New in Csound version 5.16

k-rate version fixed to documentation and SuperCollider, version 6.09

# dust2

dust2 — Random impulses.

## Description

Generates random impulses from -1 to 1.

## Syntax

```
ares dust2 kamp, kdensity
kres dust2 kamp, kdensity
```

## Performance

*kamp* -- amplitude.

*kdensity* -- average number of impulses per second.

## Examples

Here is an example of the dust2 opcode. It uses the file *dust2.csd* [examples/dust2.csd].

### Example 235. Example of the dust2 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o oscil.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kdens expon 2, p3, 20000
aout dust2 0.5, kdens
outs aout, aout

endin
</CsInstruments>
<CsScore>
i1 0 10
e
</CsScore>
```

`</CsoundSynthesizer>`

## See Also

*dust gausstrig mpulse*

## Credits

Based on James McCartney's Dust2 (SuperCollider)

Author: Tito Latini

January 2012

New in Csound version 5.16

k-rate version fixed to documentation and SuperCollider, version 6.09

## else

else — Executes a block of code when an "if...then" condition is false.

## Description

Executes a block of code when an "if...then" condition is false.

## Syntax

```
else
```

## Performance

*else* is used inside of a block of code between the "*if...then*" and *endif* opcodes. It defines which statements are executed when a "if...then" condition is false. Only one *else* statement may occur and it must be the last conditional statement before the *endif* opcode.

## Examples

Here is an example of the else opcode. It uses the file *else.csd* [examples/else.csd].

### Example 236. Example of the else opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o else.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ipch = cpspch(p4)
ienv = p5

if (ienv == 0) then
  kenv adsr 0.01, 0.95, .7, .5
else
  kenv linseg 0, p3 * .5, 1, p3 * .5, 0
endif

aout vco2      .8, ipch, 10
aout moogvcf aout, ipch + (kenv * 6 * ipch) , .5
```

```
    aout = aout * kenv
      outs aout, aout

  endin
</CsInstruments>
<CsScore>

  i 1 0 2 8.00 0
  i 1 3 2 8.00 1

  e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*elseif, endif, goto, if, igoto, kgoto, tigoto, timeout*

More information on this opcode: <http://www.csoundjournal.com/2006spring/controlFlow.html> , written by Steven Yi

## Credits

New in version 4.21

# elseif

elseif — Defines another "if...then" condition when a "if...then" condition is false.

## Description

Defines another "if...then" condition when a "if...then" condition is false.

## Syntax

```
elseif label R xb then
```

where *label* is in the same instrument block and is not an expression, and where *R* is one of the Relational operators (<, =, <=, ==, !=) (and = for convenience, see also under *Conditional Values*).

## Performance

*elseif* is used inside of a block of code between the "*if...then*" and *endif* opcodes. When a "if...then" condition is false, it defines another "if...then" condition to be met. Any number of *elseif* statements are allowed.

## Examples

Here is an example of the elseif opcode. It uses the file *elseif.csd* [examples/elseif.csd].

### Example 237. Example of the elseif opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o elseif.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ipch = cpspch(p4)
ienv = p5

if (ienv == 0) then
;ADSR
kenv adsr 0.05, 0.05, .95, .05
elseif (ienv == 1) then
;Linear Triangular Envelope
```

```
    kenv linseg 0, p3 * .5, 1, p3 * .5, 0
elseif (ienv == 2) then
    ;Ramp Up
    kenv linseg 0, p3 - .01, 1, .01, 0
endif

out vco2 .8, ipch, 10
out moogvcf aout, ipch + (kenv * 5 * ipch) , .5

aout = aout * kenv

outs aout, aout
endin
</CsInstruments>
<CsScore>

i 1 0 2 8.00 0
i 1 3 2 8.00 1
i 1 6 2 8.00 2
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*else, endif, goto, if, igoto, kgoto, tigoto, timeout*

More information on this opcode: <http://www.csoundjournal.com/2006spring/controlFlow.html> , written by Steven Yi

## Credits

New in version 4.21

# endif

endif — Closes a block of code that begins with an "if...then" statement.

## Description

Closes a block of code that begins with an "if...then" statement.

## Syntax

```
endif
```

## Performance

Any block of code that begins with an "if...then" statement must end with an *endif* statement.

## Examples

Here is an example of the endif opcode. It uses the file *endif.csd* [examples/endif.csd].

### Example 238. Example of the endif opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o endif.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
; Get the note value from the fourth p-field.
knote = p4

; Does the user want a low note?
if (knote == 0) then
  kcps = 220
; Does the user want a middle note?
elseif (knote == 1) then
  kcps = 440
; Does the user want a high note?
elseif (knote == 2) then
  kcps = 880
endif

; Create the note.
```

```
kamp init .8
ifn = 1
a1 oscili kamp, kcps, ifn

outs a1, a1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; p4: 0=low note, 1=middle note, 2=high note.
; Play Instrument #1 for one second, low note.
i 1 0 1 0
; Play Instrument #1 for one second, middle note.
i 1 1 1 1
; Play Instrument #1 for one second, high note.
i 1 2 1 2
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*elseif, else, goto, if, igoto, kgoto, tigoto, timeout*

More information on this opcode: <http://www.csoundjournal.com/2006spring/controlFlow.html> , written by Steven Yi

## Credits

New in version 4.21

# endin

endin — Ends the current instrument block.

## Description

Ends the current instrument block.

## Syntax

```
endin
```

## Initialization

Ends the current instrument block.

Instruments can be defined in any order (but they will always be both initialized and performed in ascending instrument number order). Instrument blocks cannot be nested (i.e. one block cannot contain another).



### Note

There may be any number of instrument blocks in an orchestra.

## Examples

Here is an example of the endin opcode. It uses the file *endin.csd* [examples/endin.csd].

### Example 239. Example of the endin opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o endin.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  iamp = 10000
  icps = 440
  iphs = 0
```

```
a1 oscils iamp, icps, iphs
out a1
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for 2 seconds.
i 1 0 2
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*instr*

## Credits

Example written by Kevin Conder.

# endop

endop — Marks the end of an user-defined opcode block.

## Description

Marks the end of an user-defined opcode block.

## Syntax

```
endop
```

## Performance

The syntax of a user-defined opcode block is as follows:

```
opcode name, outtypes, intypes
xinarg1 [, xinarg2] [, xinarg3] ... [xinargN] xin
[setksmps ikmps]
... the rest of the instrument's code.
xout xoutarg1 [, xoutarg2] [, xoutarg3] ... [xoutargN]
endop
```

The new opcode can then be used with the usual syntax:

```
[xinarg1] [, xinarg2] ... [xinargN] name [xoutarg1] [, xoutarg2] ... [xoutargN] [, ikmps]
```

## Examples

Here is an example of the endop opcode. It uses the file *endop.csd* [examples/endop.csd].

### Example 240. Example of the endop opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o endop.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

/* example opcode 1: simple oscillator */

opcode Oscillator, a, kk
```

```

kamp, kcps      xin           ; read input parameters
a1      vco2 kamp, kcps      ; sawtooth oscillator
        xout a1             ; write output

        endop

/* example opcode 2: lowpass filter with local ksmps */

        opcode Lowpass, a, akk

        setksmps 1           ; need sr=kr
ain, ka1, ka2   xin           ; read input parameters
aout          init 0         ; initialize output
aout          = ain*ka1 + aout*ka2 ; simple tone-like filter
        xout aout           ; write output

        endop

/* example opcode 3: recursive call */

        opcode RecursiveLowpass, a, akkpp

ain, ka1, ka2, idep, icnt     xin           ; read input parameters
        if (icnt >= idep) goto skip1      ; check if max depth reached
ain      RecursiveLowpass ain, ka1, ka2, idep, icnt + 1
skip1:
aout     Lowpass ain, ka1, ka2         ; call filter
        xout aout                     ; write output

        endop

/* example opcode 4: de-click envelope */

        opcode DeClick, a, a

ain      xin
aenv     linseg 0, 0.02, 1, p3 - 0.05, 1, 0.02, 0, 0.01, 0
        xout ain * aenv               ; apply envelope and write output

        endop

/* instr 1 uses the example opcodes */

        instr 1

kamp     = .6                       ; amplitude
kcps     expon 50, p3, 500           ; pitch
a1       oscillator kamp, kcps       ; call oscillator
kflt     linseg 0.4, 1.5, 0.4, 1, 0.8, 1.5, 0.8 ; filter envelope
a1       RecursiveLowpass a1, kflt, 1 - kflt, 10 ; 10th order lowpass
a1       DeClick a1
        outs a1, a1

        endin
</CsInstruments>
<CsScore>

i 1 0 4
e

</CsScore>
</CsoundSynthesizer>

```

## See Also

*opcode, setksmps, xin, xout*

More information on this opcode: [http://www.csoundjournal.com/2006summer/controlFlow\\_part2.html](http://www.csoundjournal.com/2006summer/controlFlow_part2.html) ,  
written by Steven Yi

The user-defined opcode page: <http://www.csounds.com/udo/> , maintained by Steven Yi

## Credits

Author: Istvan Varga, 2002; based on code by Matt J. Ingalls

New in version 4.22

# envlpx

envlpx — Applies an envelope consisting of 3 segments.

## Description

*envlpx* -- apply an envelope consisting of 3 segments:

1. stored function rise shape
2. modified exponential pseudo steady state
3. exponential decay

## Syntax

```
ares envlpx xamp, irise, idur, idec, ifn, iatss, iatdec [, ixmod]
```

```
kres envlpx kamp, irise, idur, idec, ifn, iatss, iatdec [, ixmod]
```

## Initialization

*irise* -- rise time in seconds. A zero or negative value signifies no rise modification.

*idur* -- overall duration in seconds. A zero or negative value will cause initialization to be skipped.

*idec* -- decay time in seconds. Zero means no decay. An *idec* > *idur* will cause a truncated decay.

*ifn* -- function table number of stored rise shape with extended guard point.

*iatss* -- attenuation factor, by which the last value of the *envlpx* rise is modified during the note's pseudo steady state. A factor greater than 1 causes an exponential growth and a factor less than 1 creates an exponential decay. A factor of 1 will maintain a true steady state at the last rise value. Note that this attenuation is not by fixed rate (as in a piano), but is sensitive to a note's duration. However, if *iatss* is negative (or if steady state < 4 k-periods) a fixed attenuation rate of *abs(iatss)* per second will be used. 0 is illegal.

*iatdec* -- attenuation factor by which the closing steady state value is reduced exponentially over the decay period. This value must be positive and is normally of the order of .01. A large or excessively small value is apt to produce a cutoff which is audible. A zero or negative value is illegal.

*ixmod* (optional, between +- .9 or so) -- exponential curve modifier, influencing the steepness of the exponential trajectory during the steady state. Values less than zero will cause an accelerated growth or decay towards the target (e.g. *subito piano*). Values greater than zero will cause a retarded growth or decay. The default value is zero (unmodified exponential).

## Performance

*kamp*, *xamp* -- input amplitude signal.

Rise modifications are applied for the first *irise* seconds, and decay from time *idur* - *idec*. If these periods are separated in time there will be a steady state during which *amp* will be modified by the first exponential

pattern. If the overall duration *idur* is exceeded in performance, the final decay will continue on in the same direction, tending asymptotically to zero.

## Examples

Here is an example of the *envlpx* opcode. It uses the file *envlpx.csd* [examples/envlpx.csd].

### Example 241. Example of the *envlpx* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o envlpx.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

irise = 0.2
idec = 0.5
idur = p3 - idec

ifn = 1
iatss = p5
iatdec = 0.01

kenv envlpx .6, irise, idur, idec, ifn, iatss, iatdec
kcps = cpspch(p4)
asig vco2 kenv, kcps
;apply envlpx to the filter cut-off frequency
asig moogvcf asig, kcps + (kenv * 8 * kcps) , .5 ;the higher the pitch, the higher the filter cut-off f
outs asig, asig

endin
</CsInstruments>
<CsScore>
; a linear rising envelope
f 1 0 129 -7 0 128 1

i 1 0 2 7.00 .1
i 1 + 2 7.02 1
i 1 + 2 7.03 2
i 1 + 2 7.05 3
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*envlpxr*, *linen*, *linenr*

## Credits

Thanks goes to Luis Jure for pointing out a mistake with *iatss*.

# envlpxr

envlpxr — The *envlpx* opcode with a final release segment.

## Description

*envlpxr* is the same as *envlpx* except that the final segment is entered only on sensing a MIDI note release. The note is then extended by the decay time.

## Syntax

```
ares envlpxr xamp, irise, idec, ifn, iatss, iatdec [, ixmod] [, irind]
kres envlpxr kamp, irise, idec, ifn, iatss, iatdec [, ixmod] [, irind]
```

## Initialization

*irise* -- rise time in seconds. A zero or negative value signifies no rise modification.

*idec* -- decay time in seconds. Zero means no decay.

*ifn* -- function table number of stored rise shape with extended guard point.

*iatss* -- attenuation factor, by which the last value of the *envlpxr* rise is modified during the note's pseudo steady state. A factor greater than 1 causes an exponential growth and a factor less than 1 creates an exponential decay. A factor of 1 will maintain a true steady state at the last rise value. Note that this attenuation is not by fixed rate (as in a piano), but is sensitive to a note's duration. However, if *iatss* is negative (or if steady state < 4 k-periods) a fixed attenuation rate of *abs(iatss)* per second will be used. 0 is illegal.

*iatdec* -- attenuation factor by which the closing steady state value is reduced exponentially over the decay period. This value must be positive and is normally of the order of .01. A large or excessively small value is apt to produce a cutoff which is audible. A zero or negative value is illegal.

*ixmod* (optional, between +- .9 or so) -- exponential curve modifier, influencing the steepness of the exponential trajectory during the steady state. Values less than zero will cause an accelerated growth or decay towards the target (e.g. *subito piano*). Values greater than zero will cause a retarded growth or decay. The default value is zero (unmodified exponential).

*irind* (optional) -- independence flag. If left zero, the release time (*idec*) will influence the extended life of the current note following a note-off. If non-zero, the *idec* time is quite independent of the note extension (see below). The default value is 0.

## Performance

*kamp*, *xamp* -- input amplitude signal.

*envlpxr* is an example of the special Csound “r” units that contain a note-off sensor and release time extender. When each senses a score event termination or a MIDI noteoff, it will immediately extend the performance time of the current instrument by *idec* seconds unless it is made independent by *irind*. Then it will begin a decay from wherever it was at the time.

You can use other pre-made envelopes which start a release segment upon receiving a note off message, like *linsegr* and *expsegr*, or you can construct more complex envelopes using *xtratim* and *release*. Note that you don't need to use *xtratim* if you are using *envlpxr*, since the time is extended automatically.

These “r” units can also be modified by MIDI noteoff velocities (see *veloffs*). If the *irind* flag is on (non-zero), the overall performance time is unaffected by note-off and *veloff* data.

**Multiple “r” units.** When two or more “r” units occur in the same instrument it is usual to have only one of them influence the overall note duration. This is normally the master amplitude unit. Other units controlling, say, filter motion can still be sensitive to note-off commands while not affecting the duration by making them independent (*irind* non-zero). Depending on their own *idec* (release time) values, independent “r” units may or may not reach their final destinations before the instrument terminates. If they do, they will simply hold their target values until termination. If two or more “r” units are simultaneously master, note extension is by the greatest *idec*.

## Examples

Here is an example of the *envlpxr* opcode. It uses the file *envlpxr.csd* [examples/envlpxr.csd].

### Example 242. Example of the *envlpxr* opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0   ;;realtime audio out and realtime midi in
-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o envlpxr.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

icps cpsmidi
iamp ampmidi .5

kenv envlpxr iamp, 0.2, 1, 1, 1, 0.01
asig pluck kenv, icps, 200, 2, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 129 -7 0 128 1
f 2 0 4096 10 1

f0 30 ;runs 30 seconds
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*envlpx*, *linen*, *linenr*

## Credits

Thanks goes to Luis Jure for pointing out a mistake with *iatss*.

# ephasor

ephasor — produces two outputs: a periodic phase signal and a periodic exponential decaying signal.

## Description

This opcode produces two outputs: a periodic phase signal (like the phasor opcode), and a periodic exponential decaying signal. The latter is synchronised to the former, starting at 1 and then decreasing at the same time as the phase signal increases from 0 to 1. The rate of exponential decay can be controlled by the second parameter.

## Syntax

```
aexp,aph ephasor kfreq, kR
```

## Performance

*kfreq* - the rate at which the phase and exponential signals are generated

*kR* - a parameter controlling the decay rate of the exponential signal,  $0 < kR < 1$ . Lower values produce faster decays.

## Examples

Here is an example of the ephasor opcode. It uses the file *ephasor.csd* [examples/ephasor.csd].

### Example 243. Example of the ephasor opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>

sr=44100
ksmps=64
nchnls=2
0dbfs = 1

instr 1

iamp = p4
ifr = p5
kfr expon ifr,p3,2*ifr

kfc expon 2000,p3, 4000
kq = 10
kbw = kfc/kq
kR = 1 - $M_PI*(kbw/sr)

k1 = kfc/kfr
kn = int(k1)
k1 = k1 - kn
```

```
amod,aph ephasor kfr,kR
aosc1 table aph*kn,-1,1,0,1
aosc2 table aph*(kn+1),-1,1,0,1

asig = iamp*(aosc1*(1 - k1) + aosc2*k1)*amod
outs asig, asig

endin

</CsInstruments>
<CsScore>

i1 0 10 0.5 220

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
2008

New in version 5.10

# eqfil

eqfil — Equalizer filter

## Description

The opcode eqfil is a 2nd order tunable equalisation filter based on Regalia and Mitra design ("Tunable Digital Frequency Response Equalization Filters", IEEE Trans. on Ac., Sp. and Sig Proc., 35 (1), 1987). It provides a peak/notch filter for building parametric/graphic equalisers.

The amplitude response for this filter will be flat (=1) for *kgain*=1. With *kgain* bigger than 1, there will be a peak at the centre frequency, whose width is given by the *kbw* parameter, but outside this band, the response will tend towards 1. Conversely, if *kgain* is smaller than 1, a notch will be created around the CF.

## Syntax

```
asig eqfil ain, kcf, kbw, kgain[, istor]
```

## Initialization

*istor* --initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*asig* -- filtered output signal.

*ain* -- input signal.

*kcf* -- filter centre frequency.

*kbw* -- peak/notch bandwidth (Hz).

*kgain* -- peak/notch gain.

## Examples

Here is an example of the eqfil opcode. It uses the file *eqfil.csd* [examples/eqfil.csd].

### Example 244. Example of the eqfil opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o eqfil.wav -W ;; for file output any platform
```

```
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kcf = p4
kfe expseg 10, p3*0.9, 1800, p3*0.1, 175
kenv linen .03, 0.05, p3, 0.05 ;low amplitude is needed to avoid clipping
asig buzz kenv, kfe, sr/(2*kfe), 1
afil eqfil asig, kcf, 200, 10
outs afil*20, afil*20

endin
</CsInstruments>
<CsScore>
; a sine wave.
f 1 0 16384 10 1

i 1 0 10 200 ;filter centre freq=200
i 1 + 10 1500 ;filter centre freq=1500
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
April 2007

New in version 5.06

# evalstr

evalstr — Evalstrs evaluates a string containing Csound code, returning a value.

## Description

Evalstr compiles and runs Csound code and returns a value from the global space (instr 0). This opcode can be also used to compile new instruments (as compilestr).

## Syntax

```
ires evalstr Scode  
kres evalstr Scode, ktrig
```

## Initialization

“*Scode*” -- a string to be compiled and evaluated.

## Performance

“*ktrig*” -- triggers the compilation/evaluation if non-zero.

## Examples

Here is an example of the evalstr opcode in conjunction with return:

### Example 245.

```
ival evalstr "return 2 + 2"  
print ival
```

## Credits

Author: Victor Lazzarini, 2013

# event

event — Generates a score event from an instrument.

## Description

Generates a score event from an instrument.

## Syntax

```
event "scorechar", kinsnum, kdelay, kdur, [, kp4] [, kp5] [, ...]
```

```
event "scorechar", "insname", kdelay, kdur, [, kp4] [, kp5] [, ...]
```

## Initialization

*“scorechar”* -- A string (in double-quotes) representing the first p-field in a score statement. This is usually *“e”*, *“f”*, or *“i”*.

*“insname”* -- A string (in double-quotes) representing a named instrument.

## Performance

*kinsnum* -- The instrument to use for the event. This corresponds to the first p-field, p1, in a score statement.

*kdelay* -- When (in seconds) the event will occur from the current performance time. This corresponds to the second p-field, p2, in a score statement.

*kdur* -- How long (in seconds) the event will happen. This corresponds to the third p-field, p3, in a score statement.

*kp4*, *kp5*, ... (optional) -- Parameters representing additional p-field in a score statement. It starts with the fourth p-field, p4.



### Note

Note that the *event* opcode cannot accept string p-fields. If you need to pass strings when instantiating an instrument, use the *scoreline* or *scoreline\_i* opcode.

## Examples

Here is an example of the event opcode. It uses the file *event.csd* [examples/event.csd].

### Example 246. Example of the event opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform
```

```

; Audio out   Audio in   No messages
-odac         -iadc     -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o event.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - an oscillator with a high note.
instr 1
; Create a trigger and set its initial value to 1.
ktrigger init 1

; If the trigger is equal to 0, continue playing.
; If not, schedule another event.
if (ktrigger == 0) goto contin
; kscoreop="i", an i-statement.
; kinsnum=2, play Instrument #2.
; kwhen=1, start at 1 second.
; kdur=0.5, play for a half-second.
event "i", 2, 1, 0.5

; Make sure the event isn't triggered again.
ktrigger = 0

contin:
a1 oscils 10000, 440, 1
out a1
endin

; Instrument #2 - an oscillator with a low note.
instr 2
a1 oscils 10000, 220, 1
out a1
endin

</CsInstruments>
<CsScore>

; Make sure the score plays for two seconds.
f 0 2

; Play Instrument #1 for a half-second.
i 1 0 0.5
e

</CsScore>
</CsoundSynthesizer>

```

Here is an example of the event opcode using a named instrument. It uses the file *event\_named.csd* [examples/event\_named.csd].

### Example 247. Example of the event opcode using a named instrument.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in   No messages

```

```
-odac          -iadc      -d          ;;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o event_named.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - an oscillator with a high note.
instr 1
; Create a trigger and set its initial value to 1.
ktrigger init 1

; If the trigger is equal to 0, continue playing.
; If not, schedule another event.
if (ktrigger == 0) goto contin
; kscoreop="i", an i-statement.
; kinsnum="low_note", instrument named "low_note".
; kwhen=1, start at 1 second.
; kdur=0.5, play for a half-second.
event "i", "low_note", 1, 0.5

; Make sure the event isn't triggered again.
ktrigger = 0

contin:
a1 oscils 10000, 440, 1
out a1
endin

; Instrument "low_note" - an oscillator with a low note.
instr low_note
a1 oscils 10000, 220, 1
out a1
endin

</CsInstruments>
<CsScore>

; Make sure the score plays for two seconds.
f 0 2

; Play Instrument #1 for a half-second.
i 1 0 0.5
e

</CsScore>
</CsoundSynthesizer>
```

## See also

*event\_i*, *schedule*, *schedwhen*, *schedkwhen*, *schedkwhennamed*, *scoreline*, *scoreline\_i*

## Credits

Examples written by Kevin Conder.

New in version 4.17

Thanks goes to Matt Ingalls for helping to fix the example.

Thanks goes to Matt Ingalls for helping clarify the kwhen/kdelay parameter.

# event\_i

event\_i — Generates a score event from an instrument.

## Description

Generates a score event from an instrument.

## Syntax

```
event_i "scorechar", iinsnum, idelay, idur, [, ip4] [, ip5] [, ...]
```

```
event_i "scorechar", "insname", idelay, idur, [, ip4] [, ip5] [, ...]
```

## Initialization

*“scorechar”* -- A string (in double-quotes) representing the first p-field in a score statement. This is usually *“e”*, *“f”*, or *“i”*.

*“insname”* -- A string (in double-quotes) representing a named instrument.

*iinsnum* -- The instrument to use for the event. This corresponds to the first p-field, p1, in a score statement.

*idelay* -- When (in seconds) the event will occur from the current performance time. This corresponds to the second p-field, p2, in a score statement.

*idur* -- How long (in seconds) the event will happen. This corresponds to the third p-field, p3, in a score statement.

*ip4, ip5, ...* (optional) -- Parameters representing additional p-field in a score statement. It starts with the fourth p-field, p4.

## Performance

The event is added to the queue at initialisation time.



### Note

Note that the *event\_i* opcode cannot accept string p-fields. If you need to pass strings when instantiating an instrument, use the *scoreline* or *scoreline\_i* opcode.

## Examples

Here is an example of the event\_i opcode. It uses the file *event\_i.csd* [examples/event\_i.csd].

### Example 248. Example of the event\_i opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
```

```
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o event_i.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

seed 0
gisine ftgen 0, 0, 2^10, 10, 1

instr 1 ;master instrument

ininstr = 10 ;number of called instances
indx = 0
loop:
ipan random 0, 1
ifreq random 100, 1000
iamp = 1/ininstr
event_i "i", 10, 0, p3, iamp, ifreq, ipan
loop_lt indx, 1, ininstr, loop

endin

instr 10

    print p4, p5, p6
ipeak random 0, 1 ;where is the envelope peak
asig poscil3 p4, p5, gisine
aenv transeg 0, p3*ipeak, 6, 1, p3-p3*ipeak, -6, 0
aL,aR pan2 asig*aenv, p6
outs aL, aR

endin

</CsInstruments>
<CsScore>
i1 0 10
i1 8 10
i1 16 15
e
</CsScore>
</CsoundSynthesizer>
```

## See also

*event, schedule, schedwhen, schedkwhen, schedkwhennamed, scoreline, scoreline\_i*

## Credits

Written by Istvan Varga.

New in Csound5

# exciter

exciter — A non-linear filter system to excite the signal.

## Description

"Filtered distortion to add brilliance to a signal"

## Syntax

```
ares exciter asig, kfreq, kceil, kharmonics, kblend
```

## Initialization

## Performance

*asig* -- input signal

*kfreq* -- the lower end of the harmonics created.

*kceil* -- the upper end of the harmonics created.

*kharmonics* -- amount of harmonics in the range 0.1 - 10.

*kblend* -- blend between 2nd and 3rd order harmonics in the range -10 - +10.

*exciter* is a reimplementation of the calf exciter plugin.

## Examples

Here is an example of the exciter opcode. It uses the file *exciter.csd* [examples/exciter.csd].

### Example 249. Example of the exciter opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;real-time audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o exciter.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr=44100
ksmps=32
nchnls=2
0dbfs =1

instr 1 ; excited sound + original sound
```

```
a1 diskin2 "fox.wav", 1
a2 exciter a1, 3000, 20000, 10, 10 ;generate uneven harmonics at maximum setting
  outs a2+a1, a2+a1
endin

instr 2 ; original sound for comparison

a1 diskin2 "fox.wav", 1
  outs a1, a1
endin

instr 3 ; the effect of the excited sound only

a1 diskin2 "fox.wav", 1
a2 exciter a1, 3000, 20000, 10, 10 ;generate uneven harmonics at maximum setting
  outs a2, a2
endin

</CsInstruments>
<CsScore>
i1 0 3
i2 3 3
i3 6 3
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitich after Markus Schmidt  
August 2014

New in Csound version 6.04

# exitnow

exitnow — Exit Csound as fast as possible, with no cleaning up.

## Description

In Csound4 calls an exit function to leave Csound as fast as possible. On Csound5 exits back to the driving code.

## Syntax

```
exitnow [ivalue]
```

## Initialisation

Stops Csound on the *initialisation* cycle, returning the result ivalue, which defaults to zero. Note that it is usual for this opcode to be alone in an instrument.

## Examples

Here is an example of the exitnow opcode. It uses the file *exitnow.csd* [examples/exitnow.csd].

### Example 250. Example of the exitnow opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n          ;;no sound
;-iadc      ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o exitnow.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

;after an example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

FLcolor 200, 200, 200, 0, 0, 0
; LABEL      | WIDTH | HEIGHT | X | Y
FLpanel "rtclock", 500, 130, 0, 0
;
; ON,OFF,TYPE,WIDTH, HEIGHT, X, Y, OPCODE, INS,START,IDUR
gkOnOff,ihOnOff FLbutton "On/Off", 1, 0, 22, 150, 25, 5, 5, 0, 1, 0, 3600
gkExit,ihExit FLbutton "exitnow",1, 0, 21, 150, 25, 345, 5, 0, 999, 0, 0.001
FLsetColor2 255, 0, 50, ihOnOff ;reddish color

;VALUE DISPLAY BOXES WIDTH,HEIGHT,X, Y
gidclock FLvalue "clock", 100, 25, 200, 60
FLsetVal_i 1, ihOnOff
FLpanel_end
```

```
FLrun

instr 1

if gkOnOff !=0 kgoto CONTINUE ;sense if FLTK on/off switch is not off (in which case skip the next line
turnoff          ;turn this instr. off now
CONTINUE:
ktime rtclock          ;clock continues to run even
FLprintk2 ktime, gidclock ;after the on/off button was used to stop

endin

instr 999

exitnow          ;exit Csound as fast as possible

endin
</CsInstruments>
<CsScore>

f 0 60 ;runs 60 seconds
e
</CsScore>
</CsoundSynthesizer>
```

# exp

exp — Returns e raised to the x-th power.

## Description

Returns e raised to the .xth power.

## Syntax

**exp**(x) (no rate  
restriction)

**exp**(k/i[]) (k- or i-arrays)

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

## Examples

Here is an example of the exp opcode. It uses the file *exp.csd* [examples/exp.csd].

### Example 251. Example of the exp opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o exp.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gisine   ftgen      0, 0, 2^10, 10, 1 ;table for a sine wave

instr 1 ;master instrument
koct    linseg     6, p3, 12 ; octave register straight rising from 6 to 12
kexp    linseg     0, p3/3, 5, p3/3, 5, p3/3, 0 ;exponent goes from 0 to 5 and back
kdens   =          exp(kexp) ;density is e to the power of kexp
        printks    "Generated events per second: %d\n", 1, kdens
ktrig   metro     kdens ;trigger single notes in kdens frequency
        if ktrig == 1 then
;call instr 10 for 1/kdens duration, .5 amplitude and koct register
        event      "i", 10, 0, 1/kdens, .5, koct
        endif
        endin

instr 10 ;performs one tone

```

```

ioct      rnd31      1, 0 ;random deviation maximum one octave plus/minus
aenv      transeg    p4, p3, -6, 0 ;fast decaying envelope for p4 amplitude
asin      poscil     aenv, cpsoct(p5+ioct), gisine ;sine for p5 octave register plus random deviation
          outs      asin, asin
endin

</CsInstruments>
<CsScore>
i 1 0 30
e
</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like this:

```

Generated events per second: 1
  rtevent:  T 0.000 TT 0.000 M: 0.00000 0.00000
new alloc for instr 10:
  rtevent:  T 0.811 TT 0.811 M: 0.48906 0.48906
new alloc for instr 10:
Generated events per second: 2
  rtevent:  T 1.387 TT 1.387 M: 0.48611 0.48611
  rtevent:  T 1.833 TT 1.833 M: 0.48421 0.48421
Generated events per second: 3
  rtevent:  T 2.198 TT 2.198 M: 0.47536 0.47536
  rtevent:  T 2.506 TT 2.506 M: 0.46530 0.46530
  rtevent:  T 2.773 TT 2.773 M: 0.44986 0.44986
Generated events per second: 4
  rtevent:  T 3.009 TT 3.009 M: 0.48096 0.48096
.....

```

## See Also

*abs, frac, int, log, log10, i, sqrt*

New in version 4.21

# expcurve

expcurve — This opcode implements a formula for generating a normalised exponential curve in range 0 - 1. It is based on the Max / MSP work of Eric Singer (c) 1994.

## Description

Generates an exponential curve in range 0 to 1 of arbitrary steepness. Steepness index equal to or lower than 1.0 will result in Not-a-Number errors and cause unstable behavior.

The formula used to calculate the curve is:

$$(\exp(x * \log(y))-1) / (y-1)$$

where x is equal to *kindex* and y is equal to *ksteepness*.

## Syntax

```
kout expcurve kindex, ksteepness
```

## Performance

*kindex* -- Index value. Expected range 0 to 1.

*ksteepness* -- Steepness of the generated curve. Values closer to 1.0 result in a straighter line while larger values steepen the curve.

*kout* -- Scaled output.

## Examples

Here is an example of the expcurve opcode. It uses the file *expcurve.csd* [examples/expcurve.csd].

### Example 252. Example of the expcurve opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  Silent
-odac          -iadc     -d      ;;realtime output
</CsOptions>
<CsInstruments>

sr = 48000
ksmps = 1000
nchnls = 2

instr 1 ; logcurve test

kmod phasor 1/p3
kout expcurve kmod, p4
```

```
prints "mod = %f out= %f\\n", 0.5, kmod, kout

  endin

/*--- ---*/
</CsInstruments>
<CsScore>

i1 0 5 2
i1 5 5 5
i1 10 5 30
i1 15 5 0.5

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*scale, gainslider, logcurve*

## Credits

Author: David Akbari  
October  
2006

# expon

expon — Trace an exponential curve between specified points.

## Description

Trace an exponential curve between specified points.

## Syntax

```
ares expon ia, idur, ib
```

```
kres expon ia, idur, ib
```

## Initialization

*ia* -- starting value. Zero is illegal for exponentials.

*ib* -- value after *idur* seconds. For exponentials, must be non-zero and must agree in sign with *ia*.

*idur* -- duration in seconds of the segment. A zero or negative value will cause all initialization to be skipped.

## Performance

These units generate control or audio signals whose values can pass through 2 specified points. The *idur* value may or may not equal the instrument's performance time: a shorter performance will truncate the specified pattern, while a longer one will cause the defined segment to continue on in the same direction.

## Examples

Here is an example of the expon opcode. It uses the file *expon.csd* [examples/expon.csd].

### Example 253. Example of the expon opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o expon.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
```

```
kpitch = p6
;choose between expon or line
if (kpitch == 0) then
  kpitch expon p4, p3, p5
elseif (kpitch == 1) then
  kpitch line p4, p3, p5
endif

asig vco2 .6, kpitch
outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 2 300 600 0 ;if p6=0 then expon is used
i 1 3 2 300 600 1 ;if p6=1 then line is used
i 1 6 2 600 1200 0
i 1 9 2 600 1200 1
i 1 12 2 1200 2400 0
i 1 15 2 1200 2400 1
i 1 18 2 2400 30 0
i 1 21 2 2400 30 1
e
</CsScore>
</CsSoundSynthesizer>
```

## See Also

*expseg, expsegr, line, linseg, linsegr*

# exprand

exprand — Exponential distribution random number generator (positive values only).

## Description

Exponential distribution random number generator (positive values only). This is an x-class noise generator.

## Syntax

```
ares exprand klambda
ires exprand klambda
kres exprand klambda
```

## Performance

*klambda* -- lambda parameter for the exponential distribution.

The probability density function of an exponential distribution is an exponential curve, whose mean is  $0.69515/\lambda$ . For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the exprand opcode. It uses the file *exprand.csd* [examples/exprand.csd].

### Example 254. Example of the exprand opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o exprand.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; every run time same values
```

```

klamda exprand 20
printk .2, klamda ; look
aout oscili 0.8, 440+klamda, 1 ; & listen
outs aout, aout
endin

instr 2 ; every run time different values

seed 0
klamda exprand 20
printk .2, klamda ; look
aout oscili 0.8, 440+klamda, 1 ; & listen
outs aout, aout
endin
</CsInstruments>
<CsScore>
; sine wave
f 1 0 16384 10 1

i 1 0 2
i 2 3 2
e

</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like this:

```

i 1 time 0.00033: 4.09813
i 1 time 0.20033: 56.39567
i 1 time 0.40033: 3.23362
i 1 time 0.60033: 0.24277
i 1 time 0.80033: 13.71228
i 1 time 1.00000: 12.71885
i 1 time 1.20033: 32.36737
i 1 time 1.40033: 0.29747
i 1 time 1.60033: 4.04450
i 1 time 1.80000: 35.75676
i 1 time 2.00000: 3.69845

```

Seeding from current time 3034472128

```

i 2 time 3.00033: 6.67934
i 2 time 3.20033: 2.72431
i 2 time 3.40033: 14.51822
i 2 time 3.60000: 12.10120
i 2 time 3.80033: 1.12266
i 2 time 4.00000: 26.90772
i 2 time 4.20000: 0.43554
i 2 time 4.40033: 28.59836
i 2 time 4.60033: 27.01831
i 2 time 4.80033: 18.19911
i 2 time 5.00000: 4.45125

```

## See Also

*seed, betarand, bexprnd, cauchy, gauss, linrand, pcauchy, poisson, trirand, unirand, weibull*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

# exprandi

exprandi — Exponential distribution random number generator with interpolation (positive values only).

## Description

Exponential distribution random number generator with controlled interpolation between values (positive values only). This is an x-class noise generator.

## Syntax

```
ares exprandi klambda, xamp, xcps
```

```
ires exprandi klambda, xamp, xcps
```

```
kres exprandi klambda, xamp, xcps
```

## Performance

*klambda* -- lambda parameter for the exponential distribution.

The probability density function of an exponential distribution is an exponential curve, whose mean is  $0.69515/\lambda$ . For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

*xamp* -- range over which random numbers are distributed.

*xcps* -- the frequency which new random numbers are generated.

## Examples

Here is an example of the exprandi opcode. It uses the file *exprandi.csd* [examples/exprandi.csd].

### Example 255. Example of the exprandi opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o exprand.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
```

```
nchnls = 2
odbfs = 1

instr 1
klambda exprandi 20, 1, 3
  printk2 klambda ; look
aout oscili 0.8, 440+klambda, 1 ; & listen
  outs aout, aout
endin

</CsInstruments>
<CsScore>
; sine wave
f 1 0 16384 10 1

i 1 0 4
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*exprand*

## Credits

Author: John ffitth  
Bath  
May 2011  
New in version 5.14

## expseg

expseg — Trace a series of exponential segments between specified points.

### Description

Trace a series of exponential segments between specified points.

### Syntax

```
ares expseg ia, idur1, ib [, idur2] [, ic] [...]  
kres expseg ia, idur1, ib [, idur2] [, ic] [...]
```

### Initialization

*ia* -- starting value. Zero is illegal for exponentials.

*ib*, *ic*, etc. -- value after *dur1* seconds, etc. For exponentials, must be non-zero and must agree in sign with *ia*.

*idur1* -- duration in seconds of first segment. A zero or negative value will cause all initialization to be skipped.

*idur2*, *idur3*, etc. -- duration in seconds of subsequent segments. A zero or negative value will terminate the initialization process with the preceding point, permitting the last-defined line or curve to be continued indefinitely in performance. The default is zero.

### Performance

These units generate control or audio signals whose values can pass through 2 or more specified points. The sum of *dur* values may or may not equal the instrument's performance time: a shorter performance will truncate the specified pattern, while a longer one will cause the last-defined segment to continue on in the same direction.

Note that the *expseg* opcode does not operate correctly at audio rate when segments are shorter than a k-period. Try the *expsega* opcode instead.

### Examples

Here is an example of the *expseg* opcode. It uses the file *expseg.csd* [examples/expseg.csd].

#### Example 256. Example of the expseg opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<Csoptions>  
; Select audio/midi flags here according to platform  
; Audio out  Audio in  
-odac          -iadc      ;;RT audio I/O
```

```
; For Non-realtime ouput leave only the line below:
; -o expseg.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; p4 = frequency in pitch-class notation.
kcps = cpspch(p4)

; Create an amplitude envelope.
kenv expseg 0.01, p3*0.25, 1, p3*0.75, 0.01
kamp = kenv * 30000

a1 oscil kamp, kcps, 1
out a1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for a half-second, p4=8.00
i 1 0 0.5 8.00
; Play Instrument #1 for a half-second, p4=8.01
i 1 1 0.5 8.01
; Play Instrument #1 for a half-second, p4=8.02
i 1 2 0.5 8.02
; Play Instrument #1 for a half-second, p4=8.03
i 1 3 0.5 8.03
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*expon, expsega, expsegr, line, linseg, linsegr transeg*

## Credits

Author: Gabriel Maldonado

Example written by Kevin Conder.

New in Csound 3.57

# expsega

expsega — An exponential segment generator operating at a-rate.

## Description

An exponential segment generator operating at a-rate. This unit is almost identical to *expseg*, but more precise when defining segments with very short durations (i.e., in a percussive attack phase) at audio rate.

## Syntax

```
ares expsega ia, idur1, ib [, idur2] [, ic] [...]
```

## Initialization

*ia* -- starting value. Zero is illegal.

*ib*, *ic*, etc. -- value after *idur1* seconds, etc. must be non-zero and must agree in sign with *ia*.

*idur1* -- duration in seconds of first segment. A zero or negative value will cause all initialization to be skipped.

*idur2*, *idur3*, etc. -- duration in seconds of subsequent segments. A zero or negative value will terminate the initialization process with the preceding point, permitting the last defined line or curve to be continued indefinitely in performance. The default is zero.

## Performance

This unit generate audio signals whose values can pass through two or more specified points. The sum of *dur* values may or may not equal the instrument's performance time. A shorter performance will truncate the specified pattern, while a longer one will cause the last defined segment to continue on in the same direction.

## Examples

Here is an example of the expsega opcode. It uses the file *expsega.csd* [examples/expsega.csd].

### Example 257. Example of the expsega opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o expsega.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Define a short percussive amplitude envelope that
  ; goes from 0.01 to 20,000 and back.
  aenv expsega 0.01, 0.1, 20000, 0.1, 0.01

  a1 oscil aenv, 440, 1
  out a1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #1 for one second.
i 1 1 1
; Play Instrument #1 for one second.
i 1 2 1
; Play Instrument #1 for one second.
i 1 3 1
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*expseg*, *expsegr*

## Credits

Author: Gabriel Maldonado

Example written by Kevin Conder.

New in Csound 3.57

# expsegb

expsegb — Trace a series of exponential segments between specified absolute points.

## Description

Trace a series of exponential segments between specified absolute points.

## Syntax

```
ares expsegb ia, itim1, ib [, itim2] [, ic] [...]  
kres expsegb ia, itim1, ib [, itim2] [, ic] [...]
```

## Initialization

*ia* -- starting value. Zero is illegal for exponentials.

*ib*, *ic*, etc. -- value at *tim1* seconds, etc. For exponentials, must be non-zero and must agree in sign with *ia*.

*itim1* -- time in seconds of end of first segment.

*itim2*, *itim3*, etc. -- time in seconds of subsequent ends of segments.

## Performance

These units generate control or audio signals whose values can pass through 2 or more specified points. The last *tim* value may or may not equal the instrument's performance time: a shorter performance will truncate the specified pattern, while a longer one will cause the last-defined segment to continue on in the same direction.

Note that the *expsegb* opcode does not operate correctly at audio rate when segments are shorter than a k-period. Try the *expsegba* opcode instead.

## Examples

Here is an example of the *expsegb* opcode. It uses the file *expsegb.csd* [examples/expsegb.csd].

### Example 258. Example of the expsegb opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
; Audio out  Audio in  
-odac      -iadc      ;;RT audio I/O  
; For Non-realtime output leave only the line below:  
; -o expseg.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>
```

```
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; p4 = frequency in pitch-class notation.
  kcps = cpspch(p4)

  ; Create an amplitude envelope.
  kenv expsegb 0.01, p3*0.25, 1, p3, 0.01
  kamp = kenv * 30000

  a1 oscil kamp, kcps, 1
  out a1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for a half-second, p4=8.00
i 1 0 0.5 8.00
; Play Instrument #1 for a half-second, p4=8.01
i 1 1 0.5 8.01
; Play Instrument #1 for a half-second, p4=8.02
i 1 2 0.5 8.02
; Play Instrument #1 for a half-second, p4=8.03
i 1 3 0.5 8.03
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*expon, expseg, expsega, expsegr, line, linseg, linsegr transeg*

## Credits

Author: Victor Lazzarini  
June 2011

New in version 5.14

# expsegba

expsegba — An exponential segment generator operating at a-rate with absolute times.

## Description

An exponential segment generator operating at a-rate. This unit is almost identical to *expsegb*, but more precise when defining segments with very short durations (i.e., in a percussive attack phase) at audio rate.

## Syntax

```
ares expsegba ia, itim1, ib [, itim2] [, ic] [...]
```

## Initialization

*ia* -- starting value. Zero is illegal.

*ib*, *ic*, etc. -- value after *itim1* seconds, etc. must be non-zero and must agree in sign with *ia*.

*itim1* -- time in seconds at end of first segment.

*itim2*, *itim3*, etc. -- time in seconds at the end of subsequent segments.

## Performance

This unit generate audio signals whose values can pass through two or more specified points. The final *tim* value may or may not equal the instrument's performance time. A shorter performance will truncate the specified pattern, while a longer one will cause the last defined segment to continue on in the same direction.

## Examples

Here is an example of the expsegba opcode. It uses the file *expsegba.csd* [examples/expsegba.csd].

### Example 259. Example of the expsegba opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o expsega.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
```

```
nchnls = 1

; Instrument #1.
instr 1
  ; Define a short percussive amplitude envelope that
  ; goes from 0.01 to 20,000 and back.
  aenv expsegba 0.01, 0.1, 20000, 0.2, 0.01

  a1 oscil aenv, 440, 1
  out a1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #1 for one second.
i 1 1 1
; Play Instrument #1 for one second.
i 1 2 1
; Play Instrument #1 for one second.
i 1 3 1
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*expsegb, expsegr*

## Credits

Author: John fitch

June 2011

New in Csound 5.14

# expsegr

expsegr — Trace a series of exponential segments between specified points including a release segment.

## Description

Trace a series of exponential segments between specified points including a release segment.

## Syntax

```
ares expsegr ia, idur1, ib [, idur2] [, ic] [...], irel, iz
```

```
kres expsegr ia, idur1, ib [, idur2] [, ic] [...], irel, iz
```

## Initialization

*ia* -- starting value. Zero is illegal for exponentials.

*ib, ic, etc.* -- value after *dur1* seconds, etc. For exponentials, must be non-zero and must agree in sign with *ia*.

*idur1* -- duration in seconds of first segment. A zero or negative value will cause all initialization to be skipped.

*idur2, idur3, etc.* -- duration in seconds of subsequent segments. A zero or negative value will terminate the initialization process with the preceding point, permitting the last-defined line or curve to be continued indefinitely in performance. The default is zero.

*irel, iz* -- duration in seconds and final value of a note releasing segment.

## Performance

These units generate control or audio signals whose values can pass through 2 or more specified points. The sum of *dur* values may or may not equal the instrument's performance time: a shorter performance will truncate the specified pattern, while a longer one will cause the last-defined segment to continue on in the same direction.

*expsegr* is amongst the Csound “r” units that contain a note-off sensor and release time extender. When each senses an event termination or MIDI noteoff, it immediately extends the performance time of the current instrument by *irel* seconds, and sets out to reach the value *iz* by the end of that period (no matter which segment the unit is in). “r” units can also be modified by MIDI noteoff velocities. For two or more extenders in an instrument, extension is by the greatest period.

You can use other pre-made envelopes which start a release segment upon receiving a note off message, like *linsegr* and *madsr*, or you can construct more complex envelopes using *xtratim* and *release*. Note that you don't need to use *xtratim* if you are using *expsegr*, since the time is extended automatically.

## Examples

Here is an example of the *expsegr* opcode. It uses the file *expsegr.csd* [examples/expsegr.csd].

## Example 260. Example of the expsegr opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0   ;;realtime audio out and realtime midi in
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o expsegr.wav -W   ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

icps cpsmidi
iamp ampmidi .3

kenv expsegr 1, .05, 0.5, 1, .01
asig pluck kenv, icps, 200, 1, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 4096 10 1 ;sine wave

f0 30 ;runs 30 seconds
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*linsegr, expsegr, envlpxr, mxadsr, madsr expon, expseg, expsega, xtratim*

## Credits

Author: Barry L. Vercoe

New in Csound 3.47

# faustaudio

faustaudio — Instantiates and runs a compiled Faust program.

## Description

Faustaudio will instantiate and run a Faust program compiled with faustcompile.

## Syntax

```
ihandle, a1[, a2, ...] faustaudio ifac[, ain1, ...]
```

## Initialization

“ifac” -- a handle to a compiled Faust program, produced by faustcompile.

“ihandle” -- a handle to the Faust DSP instance, which can be used to access its controls with faustctl.

## Performance

“ain1,…” -- input signals

“a1,…” -- output signals

## Examples

Here is an example of the faustaudio opcode. It uses the file *faustaudio.csd* [examples/faustaudio.csd].

### Example 261. Example of the faustaudio opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
instr 1
  a1 rand 0dbfs/4
  a2 oscili 0dbfs/4, 440
  ihandle faustcompile "process=+;", "-vec -lv 1"
  idsp, asig faustaudio ihandle, a1, a2
  out asig
endin
</CsInstruments>
<CsScore>
i1 0 10
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini, 2013

# faustcompile

`faustcompile` — Invokes the just-in-time compiler to produce a instantiable DSP process from a Faust program.

## Description

Faustcompile will compile a Faust program from a string, controlled by various arguments. Multi-line strings are accepted, using `{ { }` to enclose the string.

## Syntax

```
ihandle faustcompile Scode, Sargs
```

## Initialization

“*Scode*” -- a string (in double-quotes or enclosed by `{ { }`) containing a Faust program.

“*Sargs*” -- a string (in double-quotes or enclosed by `{ { }`) containing Faust compiler args.

## Examples

Here is an example of the `faustcompile` opcode. It uses the file `faustcompile.csd` [examples/faustcompile.csd].

### Example 262. Example of the `faustcompile` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
instr 1
  a1 rand 0dbfs/4
  a2 oscili 0dbfs/4, 440
  ihandle faustcompile "process==;", "-vec -lv 1"
  idsp,asig faustaudio ihandle,a1,a2
  out asig
endin
</CsInstruments>
<CsScore>
i1 0 10
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini, 2013

# faustctl

faustctl — Adjusts a given control in a Faust DSP instance.

## Description

Faustctl will set a given control in a running faust program

## Syntax

```
faustctl idsp, Scontrol, kval
```

## Initialization

“*Scontrol*” -- a string containing the control name

“*idsp*” -- a handle to an existing Faust DSP instance

## Performance

“*kval*” -- value to which the control will be set.

## Examples

Here is an example of the faustctl opcode. It uses the file *faustctl.csd* [examples/faustctl.csd].

### Example 263. Example of the faustctl opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
instr 1
  ain1 oscili 0dbfs/2, 440
  idsp, a1 faustgen {{
    gain = hslider("vol", 1, 0, 1, 0.01);
    process = (_ * gain);
  }}, ain1
  k1 line 0, p3, 1
  faustctl idsp, "vol", k1
  out a1
endin
</CsInstruments>
<CsScore>
i1 0 10
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini, 2013

# faustgen

faustgen — Compiles, Instantiates and runs a compiled Faust program.

## Description

Faustgen will invoke the just-in-time compiler, instantiate and run a Faust program.

## Syntax

```
ihandle, a1[, a2, ...] faustgen SCode[, ain1, ...]
```

## Initialization

“Scode” -- a string containing a Faust program.

“ihandle” -- a handle to the Faust DSP instance, which can be used to access its controls with faustctl.

## Performance

“ain1,…” -- input signals

“a1,…” -- output signals

## Examples

Here is an example of the faustgen opcode. It uses the file *faustgen.csd* [examples/faustgen.csd].

### Example 264. Example of the faustgen opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
instr 1
ain1 oscili 0dbfs/2, 440
idsp, a1 faustgen {{
    gain = hslider("vol", 1, 0, 1, 0.01);
    process = (_ * gain);
}}, ain1
k1 line 0, p3, 1
faustctl idsp, "vol", k1
    out a1
endin
</CsInstruments>
<CsScore>
i1 0 10
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini, 2013

# fareylen

fareylen — returns the length of a Farey Sequence.

## Description

This opcode can be used in conjunction with *GENfarey*. It calculates the length of Farey Sequence  $F_n$ . Its length is given by:  $|F_n| = 1 + \text{SUM over } n \text{ phi}(m)$  where  $\text{phi}(m)$  is Euler's totient function, which gives the number of integers  $\leq m$  that are coprime to  $m$ .

Some values for the length of  $F_n$  given  $n$ :

<b>n</b>	<b>F<sub>n</sub></b>
1	2
2	3
3	5
4	7
5	11
6	13
7	19
8	23
9	29
10	33
11	43
12	47
13	59
14	65
15	73
16	81
17	97
18	103
19	121
20	129

## Syntax

```
kfl fareylen kfn
```

## Performance

The length of the identified Farey sequence is returned.

*kfn* -- Integer identifying the sequence.

## Credits

Author: Georg Boenn  
University of Glamorgan, UK

New in Csound version 5.13

# fareyleni

fareyleni — returns the length of a Farey Sequence.

## Description

This opcode can be used in conjunction with *GENfarey*. It calculates the length of Farey Sequence  $F_n$ . Its length is given by:  $|F_n| = 1 + \text{SUM over } n \text{ phi}(m)$  where  $\text{phi}(m)$  is Euler's totient function, which gives the number of integers  $\leq m$  that are coprime to  $m$ .

Some values for the length of  $F_n$  given  $n$ :

<b>n</b>	<b>F<sub>n</sub></b>
1	2
2	3
3	5
4	7
5	11
6	13
7	19
8	23
9	29
10	33
11	43
12	47
13	59
14	65
15	73
16	81
17	97
18	103
19	121
20	129

## Syntax

```
ifl fareyleni ifn
```

## Initialisation

The length of the identified Farey sequence is returned.

*ifn* -- Integer identifying the sequence.

## Examples

Here is an example of the fareyleni opcode. It uses the file *fareyleni.csd* [examples/fareyleni.csd].

### Example 265. Example of the fareyleni opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc for RT audio input as well
; For Non-realtime ouput leave only the line below:
; -o fareyleni.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

; initialise integer n for Farey Sequence F_8
gifarn init 8

; calculate length of F_8, it should return 23 for |F_8|
gires fareyleni gifarn

; convert to negative number for the GEN routine because
; the table length won't be a power of 2
; (The length of a Farey Sequence of n > 1 is always an odd number)
gilen init gires * -1

; create F_8 with GENfarey, the negative table number prevents
; unnecessary normalisation (F_8 IS already normalised)
;          n      mode:
gifarey ftgen 100, 0, gilen, "farey", gifarn, 4
; if mode=4 then 1 is added to each element of F_n.
; Useful for creating just tuning tables that can be read by the cps2pch opcode.

instr 1
; the very last element of F_n is not needed in the case of tuning tables
ires = gires - 1
; read out and print to file
kndx init 0
if (kndx < ires) then
    kelem tab kndx, gifarey
    fprintks "farey8_tuning.txt", "%2.6f\\n", kelem
    kndx = kndx+1
endif

endin

instr 2

ip  cps2pch p4, -100
asig poscil .5, ip, 1
aenv linseg 0, 0.1, 1, p3-0.2, 1, 0.1, 0
outs asig * aenv, asig * aenv

endin
</CsInstruments>

```



# ficlose

ficlose — Closes a previously opened file.

## Description

*ficlose* can be used to close a file which was opened with *fiopen*.

## Syntax

```
ficlose ihandle
```

```
ficlose Sfilename
```

## Initialization

*ihandle* -- a number which identifies this file (generated by a previous *fiopen*).

*Sfilename* -- A string in double quotes or string variable with the filename. The full path must be given if the file directory is not in the system PATH and is not present in the current directory.

## Performance

*ficlose* closes a file which was previously opened with *fiopen*. *ficlose* is only needed if you need to read a file written to during the same csound performance, since only when csound ends a performance does it close and save data in all open files. The opcode *ficlose* is useful for instance if you want to save presets within files which you want to be accesible without having to terminate csound.



### Note

If you don't need this functionality it is safer not to call *ficlose*, and just let csound close the files when it exits.

If a files closed with *ficlose* is being accessed by another opcode (like *fout* or *foutk*), it will be closed later when it is no longer being used.



### Warning

This opcode should be used with care, as the file handle will become invalid, and will cause an init error when an opcode tries to access the closed file.

## Examples

Here is an example of the ficlose opcode. It uses the file *ficlose.csd* [examples/ficlose.csd].

### Example 266. Example of the ficlose opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o ficlose.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gihand fiopen "test1.txt", 0

instr 1

ires random 0, 100
fouti gihand, 0, 1, ires
ficlose gihand

endin
</CsInstruments>
<CsScore>

i 1 0 1

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*fiopen, fout, fouti, foutir, foutk*

## Credits

Author: Gabriel Maldonado  
Italy  
1999

New in Csound version 5.02

# filebit

filebit — Returns the number of bits in each sample in a sound file.

## Description

Returns the number of bits in each sample in a sound file.

## Syntax

```
ir filebit ifilcod [, iallowraw]
```

## Initialization

*ifilcod* -- sound file to be queried

*iallowraw* -- (Optional) Allow raw sound files (default=1)

## Performance

*filebit* returns the number of bits in each sample in the sound file *ifilcod*. In the case of floating point samples the value -1 is returned for floats and -2 for doubles. For non-PCM formats the value is negative, and based on libsndfile's format encoding.

## Examples

Here is an example of the filebit opcode. It uses the file *filebit.csd* [examples/filebit.csd], and *mary.wav* [examples/mary.wav].

### Example 267. Example of the filebit opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o filebit.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Print out the number of channels in the
; audio file "mary.wav".
```

```
    ibits filebit "mary.wav"
    print ibits
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for 1 second.
i 1 0 1
e

</CsScore>
</CsoundSynthesizer>
```

The audio file “mary.wav” is in monoaural CD format, so *filebit*'s output should include a line like this:

```
instr 1:  ibits = 16.000
```

## See Also

*filelen, filenchnls, filepeak, filesr*

## Credits

Author: Victor Lazzarini  
July 1999

Example written by John ffitth.

New in Csound version 5.11

# filelen

filelen — Returns the length of a sound file.

## Description

Returns the length of a sound file.

## Syntax

```
ir filelen ifilcod, [iallowraw]
```

## Initialization

*ifilcod* -- sound file to be queried

*iallowraw* -- Allow raw sound files (default=1)

## Performance

*filelen* returns the length of the sound file *ifilcod* in seconds. *filelen* can return the length of convolve and PVOC files if the "allow raw sound file" flag is not zero (it is non-zero by default).

## Examples

Here is an example of the *filelen* opcode. It uses the file *filelen.csd* [examples/filelen.csd], *fox.wav* [examples/fox.wav], and *kickroll.wav* [examples/kickroll.wav].

### Example 268. Example of the *filelen* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
;-odac      ;;realtime audio out
-iadc      ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
;-o filelen.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; choose between mono or stereo file

ilen  filelen p4 ;calculate length of soundfile
print ilen
ichn  filechnls p4 ;check number of channels
;print ichn
```

```
if (ichn == 1) then
;mono signal
asig diskin2 p4, 1
outs asig, asig

else
;stereo signal
aL, aR diskin2 p4, .5, 0, 1
outs aL, aR

endif

endin
</CsInstruments>
<CsScore>

i 1 0 3 "fox.wav" ;mono signal
i 1 5 2 "kickroll.wav" ;stereo signal

e
</CsScore>
</CsoundSynthesizer>
```

The mono audio file “fox.wav” is 2.8 seconds long, and the stereo file “kickroll.wav” is 0.9 seconds. So *filelen*'s output should include a line for the mono and the stereo file like this:

```
instr 1: ilen = 2.757
instr 1: ilen = 0.857
```

## See Also

*filebit, filenchnls, filepeak, filesr*

## Credits

Author: Matt Ingalls  
July 1999

New in Csound version 3.57

# filenchnls

filenchnls — Returns the number of channels in a sound file.

## Description

Returns the number of channels in a sound file.

## Syntax

```
ir filenchnls ifilcod [, iallowraw]
```

## Initialization

*ifilcod* -- sound file to be queried

*iallowraw* -- (Optional) Allow raw sound files (default=1)

## Performance

*filenchnls* returns the number of channels in the sound file *ifilcod*. *filechnls* can return the number of channels of convolve and PVOC files if the *iallowraw* flag is not zero (it is non-zero by default).

## Examples

Here is an example of the *filenchnls* opcode. It uses the file *filenchnls.csd*, [examples/filenchnls.csd] *fox.wav* [examples/fox.wav], and *kickroll.wav* [examples/kickroll.wav].

### Example 269. Example of the *filenchnls* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
;-odac      ;;realtime audio out
-iadc      ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
;-o filechnls.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; choose between mono or stereo file

ilen  filelen p4 ;calculate length of soundfile
;print ilen
ichn  filenchnls p4 ;check number of channels
print ichn
```

```
if (ichn == 1) then
;mono signal
asig diskin2 p4, 1
  outs    asig, asig

else
;stereo signal
aL, aR diskin2 p4, .5, 0, 1
  outs    aL, aR

endif

endin
</CsInstruments>
<CsScore>

i 1 0 3 "fox.wav" ;mono signal
i 1 5 2 "kickroll.wav" ;stereo signal

e
</CsScore>
</CsoundSynthesizer>
```

The audio file “fox.wav” is monoaural (1 channel), while “kickroll.wav” is stereo (2 channels) So *fi-lenchls*'s output should include lines like this:

```
instr 1:  ichn = 1.000
instr 1:  ichn = 2.000
```

## See Also

*filebit, filelen, filepeak, filesr*

## Credits

Author: Matt Ingalls  
July 1999

New in Csound version 3.57

# filepeak

filepeak — Returns the peak absolute value of a sound file.

## Description

Returns the peak absolute value of a sound file.

## Syntax

```
ir filepeak ifilcod [, ichnl]
```

## Initialization

*ifilcod* -- sound file to be queried

*ichnl* (optional, default=0) -- channel to be used in calculating the peak value. Default is 0.

- *ichnl* = 0 returns peak value of all channels
- *ichnl* > 0 returns peak value of *ichnl*

## Performance

*filepeak* returns the peak absolute value of the sound file *ifilcod*.

## Examples

Here is an example of the filepeak opcode. It uses the file *filepeak.csd* [examples/filepeak.csd], and *Church.wav* [examples/Church.wav].

### Example 270. Example of the filepeak opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o filepeak.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

iscaldb = p4 ;set peak amplitude in dB
```

```
ipeak  filepeak "Church.wav"
iscal  = ampdb(iscaldb)/ipeak ;calculate amp multiplier
printf_i "\nPeak value in file '%s' is %f (%.3f dB).\n\n", 1, "Church.wav", ipeak, dbamp(ipeak)

asnd soundin "Church.wav"
outs asnd, asnd
; scale & write file to disk
asig = asnd*iscal ;scale to p4
fout "Church_norm.wav", 14, asig

endin

instr 2 ; play scaled file

aout soundin "Church_norm.wav"
ipknew filepeak "Church_norm.wav"
printf_i "\nPeak value in file '%s' is %f (%.3f dB).\n\n", 1, "Church_norm.wav", ipknew, dbamp(ipknew)
outs aout, aout

endin
</CsInstruments>
<CsScore>

i 1 0 2 -6 ; normalize audio to -6 dB
i 2 2 2
e
</CsScore>
</CsoundSynthesizer>
```

The *filepeak*'s output should include lines like this:

```
Peak value in file 'Church.wav' is 0.909363 (-0.825 dB).
Peak value in file 'Church_norm.wav' is 0.501190 (-6.000 dB).
```

## See Also

*filelen, filenchnls, filesr*

## Credits

Author: Matt Ingalls  
July 1999

New in Csound version 3.57

# filescal

filescal — Phase-locked vocoder processing with onset detection/processing, 'tempo-scaling'.

## Description

*filescal* implements phase-locked vocoder processing from disk files, resampling if necessary.

This opcode allows for time and frequency-independent scaling. Time is advanced internally, but controlled by a tempo scaling parameter; when an onset is detected, timescaling is momentarily stopped to avoid smearing of attacks. The quality of the effect is generally improved with phase locking switched on.

*filescal* will also scale pitch, independently of frequency, using a transposition factor (k-rate).

## Syntax

```
asig[,asig2] filescal ktimescal, kamp, kpitch, Sfile, klock [,ifftsize, idecim, ithresh]
```

## Initialization

*Sfile* -- source soundfile, mono or stereo files are allowed, but need to match the number of outputs.

*ifftsize* -- FFT size (power-of-two), defaults to 2048.

*idecim* -- decimation, defaults to 4 (meaning hopsize = fftsize/4)

*idbthresh* -- threshold based on dB power spectrum ratio between two successive windows. A detected ratio above it will cancel timescaling momentarily, to avoid smearing (defaults to 1)

## Performance

*ktimescal* -- timescaling ratio, < 1 stretch, > 1 contract. Non-negative numbers only.

*kamp* -- amplitude scaling

*kpitch* -- grain pitch scaling (1=normal pitch, < 1 lower, > 1 higher; negative, backwards)

*klock* -- 0 or 1, to switch phase-locking on/off

## Examples

Here is an example of the filescal opcode. It uses the file *filescal.csd* [examples/filescal.csd].

### Example 271. Example of the filescal opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
</CsOptions>
```

```
<CsInstruments>

instr 1
p3 = filelen("fox.wav")/p4
k1 linen 1,0.01,p3,0.1
a1 filescal p4,0.5,1,"fox.wav",1
  out a1*k1
endin

</CsInstruments>
<CsScore>
i1 0 1 1.25
i1 2.5 1 .75
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
April 2016

New in version 6.07

April 2016.

# filesr

filesr — Returns the sample rate of a sound file.

## Description

Returns the sample rate of a sound file.

## Syntax

```
ir filesr ifilcod [, iallowraw]
```

## Initialization

*ifilcod* -- sound file to be queried

*iallowraw* -- (Optional) Allow raw sound files (default=1)

## Performance

*filesr* returns the sample rate of the sound file *ifilcod*. *filesr* can return the sample rate of convolve and PVOC files if the *iallowraw* flag is not zero (it is non-zero by default).

## Examples

Here is an example of the filesr opcode. It uses the file *filesr.csd* [examples/filesr.csd], and *beats.wav* [examples/beats.wav].

### Example 272. Example of the filesr opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o filesr.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

;after an example from Jonathan Murphy

instr 1
;load sound into an ftable
Sfile      strcpy      "beats.wav"
ilen       filelen   Sfile
```

```
    isr      filesr    Sfile
    isamps   = ilen * isr
    ;adjust the length of the table to be a power of two closest
    ;to the actual size of the sound
    isize    init     1
loop:
    isize    = isize * 2
    if (isize < isamps) igoto loop
    itab     ftgen    0, 0, isize, 1, Sfile, 0, 0, 0
prints "sample rate = %f, size = %f\n", isr, isize ;prints them

endin
</CsInstruments>
<CsScore>
i1 0 2
e
</CsScore>
</CsoundSynthesizer>
```

The audio file “beats.wav” was sampled at 44.1 KHz. So *filesr*'s output should include a line like this:

```
sample rate = 44100.000000, size = 131072.000000
```

## See Also

*filebit, filelen, filenchnls, filepeak*

## Credits

Author: Matt Ingalls

July 1999

New in Csound version 3.57

# filevalid

filevalid — Checks that a file can be used.

## Description

Returns 1 if the sound file is valid, or 0 if not.

## Syntax

```
ir filevalid ifilcod
```

## Initialization

*ifilcod* -- sound file to be queried

## Performance

*filevalid* returns 1 if the sound file *ifilcod* can be used.

## Examples

Here is an example of the filevalid opcode. It uses the file *filevalid.csd* [examples/filevalid.csd].

### Example 273. Example of the filevalid opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
;-o filevalid.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
Sfile      strget    p4
ivld       filevalid Sfile

if ivld>0 then
asig       diskin2   Sfile, 1
           outs      asig, asig
else
           printf_i  "Audiofile '%s' does not exist!\n", 1, Sfile
endif
endin
```

```
</CsInstruments>
<CsScore>

i 1 0 3 "frox.wav";file does not exist!!!
i 1 + 3 "fox.wav" ;but this one certainly does...

e
</CsScore>
</CsoundSynthesizer>
```

The output should include a line like this:

```
Audiofile 'frox.wav' does not exist!
```

## See Also

*filebit, filelen, filenchnls, filepeak, filesr*

## Credits

Author: Matt Ingalls  
July 2010

New in Csound version 5.13

# fillarray

fillarray — Generate a vector with initial values.

## Description

Generate a vector (one-dimensional k-rate array) with a sequence of numeric or string values.

## Syntax

```
karray[] fillarray ival1, ival2,....ivaln  
karray fillarray ival1, ival2,....ivaln
```

## Initialization

*ival1,...ivaln* -- values to place in the vector.

In the second form the answer array must be pre-declared, and it may be a multidimensional array which if filled in row-major order.

## Examples

Here is an example of the fillarray opcode. It uses the file *fillarray.csd* [examples/fillarray.csd].

### Example 274. Example of the fillarray opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
-n  
</CsOptions>  
<CsInstruments>  
  
sr = 44100  
ksmps = 32  
nchnls = 2  
0dbfs = 1  
  
instr 1  
  
  kS[] fillarray 1,7,5  
    printk 0, kS[0]  
    printk 0, kS[1]  
    printk 0, kS[2]  
  turnoff  
endin  
  
</CsInstruments>  
<CsScore>  
i 1 0 1  
e  
</CsScore>
```

`</CsoundSynthesizer>`

## See Also

Vectorial opcodes array opcodes

## Credits

Author: John ffitth  
Codemist Ltd  
2013

New in version 6.00

# fft

fft — Complex-to-complex Fast Fourier Transform.

## Description

Applies a forward Fast Fourier Transform to a complex-valued input 1-dimensional array producing a complex-valued output. The output is another array containing the complex-valued signal, and both arrays are arranged in interleaved real-imaginary format. The output array will have the same size as the input, and the transform size will be equivalent to half of the length of the array. Non-power-of-two transforms are limited to even sizes with not too many factors.

## Syntax

```
kout[] fft kin[]
```

## Performance

*kout[]* -- output array containing the complex-valued output. It will be created if it does not exist.

*kin[]* -- input array containing the complex-valued input.

## Examples

Here is an example of the `fft` opcode. It uses the file *fft.csd* [examples/fft.csd].

### Example 275. Example of the `fft` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-d -o dac
</CsOptions>
<CsInstruments>
ksmps = 64

instr 1
  ifftsize = 1024
  kcnt init 0
  kIn[] init ifftsize
  kOut[] init ifftsize

  a1 oscili 0dbfs/2, 440

  if kcnt >= ifftsize then
    kCmplx[] r2c kIn
    kSpec[] fft kCmplx
    kCmplx fftin kSpec
    kOut c2r kCmplx
    kcnt = 0
  endif

kIn[] shiftin a1
```

```
a2 shiftout kOut  
kcnt += ksmps  
  out a2  
endin  
</CsInstruments>  
<CsScore>  
i1 0 10  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

Vectorial opcodes, array opcodes

## Credits

Author: Victor Lazzarini  
NUI Maynooth  
2014

New in version 6.04

## filter2

`filter2` — Performs filtering using a transposed form-II digital filter lattice with no time-varying control.

### Description

General purpose custom filter with no time-varying pole control. The filter coefficients implement the following difference equation:

$$(1)*y(n) = b0*x[n] + b1*x[n-1] + \dots + bM*x[n-M] - a1*y[n-1] - \dots - aN*y[n-N]$$

the system function for which is represented by:

$$H(Z) = \frac{B(Z)}{A(Z)} = \frac{b0 + b1*Z^{-1} + \dots + bM*Z^{-M}}{1 + a1*Z^{-1} + \dots + aN*Z^{-N}}$$

### Syntax

```
ares filter2 asig, iM, iN, ib0, ib1, ..., ibM, ia1, ia2, ..., iaN
```

```
kres filter2 ksig, iM, iN, ib0, ib1, ..., ibM, ia1, ia2, ..., iaN
```

### Initialization

At initialization the number of zeros and poles of the filter are specified along with the corresponding zero and pole coefficients. The coefficients must be obtained by an external filter-design application such as Matlab and specified directly or loaded into a table via *GEN01*.

### Performance

The *filter2* opcodes perform filtering using a transposed form-II digital filter lattice with no time-varying control.

Since *filter2* implements generalized recursive filters, it can be used to specify a large range of general DSP algorithms. For example, a digital waveguide can be implemented for musical instrument modeling using a pair of *delayr* and *delayw* opcodes in conjunction with the *filter2* opcode.

### Examples

A first-order linear-phase lowpass FIR filter operating on a k-rate signal:

```
k1 filter2 ksig, 2, 0, 0.5, 0.5 ;; k-rate FIR filter
```

Here is another example of the *filter2* opcode. It uses the file *filter2.csd* [examples/filter2.csd].

## Example 276. Example of the filter2 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc  ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o filter2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; 2 saw waves of which one is slightly detuned

ib1 = p5
ivol = p6 ;volume to compensate
kcps init cpspch(p4)
asig1 vco2 .05, kcps ;saw 1
asaw1 filter2 asig1, 1, 1, 1, ib1 ;filter 1
asig2 vco2 .05, kcps+1 ;saw 2
asaw2 filter2 asig2, 1, 1, 1, ib1 ;filter 2
aout = (asaw1+asaw2)*ivol ;mix
outs aout, aout

endin
</CsInstruments>
<CsScore>

i 1 0 4 6.00 -.001 5 ;different filter values
i 1 + 4 6.00 -.6 2 ;and different volumes
i 1 + 4 6.00 -.95 .3 ;to compensate
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*zfilter2*

## Credits

Author: Michael A. Casey  
M.I.T.  
Cambridge, Mass.  
1997

New in version 3.47

# fin

*fin* — Read signals from a file at a-rate.

## Description

Read signals from a file at a-rate.

## Syntax

```
fin ifilename, iskipframes, iformat, ain1 [, ain2] [, ain3] [...]  
fin ifilename, iskipframes, iformat, arr[]
```

## Initialization

*ifilename* -- input file name (can be a string or a handle number generated by *fiopen*).

*iskipframes* -- number of frames to skip at the start (every frame contains a sample of each channel)

*iformat* -- a number specifying the input file format for headerless files. If a header is found, this argument is ignored.

- 0 - 32 bit floating points without header
- 1 - 16 bit integers without header

## Performance

*fin* (file input) is the complement of *fout*: it reads a multichannel file to generate audio rate signals. The user must be sure that the number of channels of the input file is the same as the number of *ainX* arguments.



### Note

Please note that since this opcode generates its output using input parameters (on the right side of the opcode), these variables must be initialized before use, otherwise a 'used before defined' error will occur. You can use the *init* opcode for this.

## Examples

Here is an example of the *fin* opcode. It uses the file *fin.csd* [examples/fin.csd] and *fox.wav* [examples/fox.wav].

### Example 277. Example of the *fin* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform
```

```
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o fin.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

asnd init 0      ;input of fin must be initialized
    fin "fox.wav", 0, 0, asnd ;read audiofile
aenv follow asnd, 0.01 ;envelope follower
kenv downsamp aenv
asig rand kenv   ;gate the noise with audiofile
outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 3
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*fini, fink*

## Credits

Author: Gabriel Maldonado  
Italy  
1999  
Author: John ffitc  
NUIM, 2013

New in Csound version 3.56

Array version new in 6.01

# fini

`fini` — Read signals from a file at *i*-rate.

## Description

Read signals from a file at *i*-rate.

## Syntax

```
fini ifilename, iskipframes, iformat, in1 [, in2] [, in3] [, ...]
```

## Initialization

*ifilename* -- input file name (can be a string or a handle number generated by *fiopen*)

*iskipframes* -- number of frames to skip at the start (every frame contains a sample of each channel)

*iformat* -- a number specifying the input file format. If a header is found, this argument is ignored.

- 0 - floating points in text format (loop; see below)
- 1 - floating points in text format (no loop; see below)
- 2 - 32 bit floating points in binary format (no loop)

## Performance

*fini* is the complement of *fouti* and *foutir*. It reads the values each time the corresponding instrument note is activated. When *iformat* is set to 0 and the end of file is reached, the file pointer is zeroed. This restarts the scan from the beginning. When *iformat* is set to 1 or 2, no looping is enabled and at the end of file the corresponding variables will be filled with zeroes.



### Note

Please note that since this opcode generates its output using input parameters (on the right side of the opcode), these variables must be initialized before use, otherwise a 'used before defined' error will occur. You can use the *init* opcode for this.

## Examples

Here is an example of the *fini* opcode. It uses the file *fini.csd* [examples/*fini.csd*].

### Example 278. Example of the *fini* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

Note that this example requires the file *test.txt* to be created, for example by *fouti*

```
<CsoundSynthesizer>
```

```
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o fini.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gihand fiopen "test.txt", 1

instr 1

  ihz  init 0
      fini gihand, 0, 1, ihz
  ar   oscil 0.5, ihz
      outs ar, ar
endin
</CsInstruments>
<CsScore>

i 1 0 1

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*fin, fink*

## Credits

Author: Gabriel Maldonado  
Italy  
1999

New in Csound version 3.56

# fink

fink — Read signals from a file at k-rate.

## Description

Read signals from a file at k-rate.

## Syntax

```
fink ifilename, iskipframes, iformat, kin1 [, kin2] [, kin3] [...]
```

## Initialization

*ifilename* -- input file name (can be a string or a handle number generated by *fiopen*)

*iskipframes* -- number of frames to skip at the start (every frame contains a sample of each channel)

*iformat* -- a number specifying the input file format. If a header is found, this argument is ignored.

- 0 - 32 bit floating points without header
- 1 - 16 bit integers without header

## Performance

*fink* is the same as *fin* but operates at k-rate.



### Note

Please note that since this opcode generates its output using input parameters (on the right side of the opcode), these variables must be initialized before use, otherwise a 'used before defined' error will occur. You can use the *init* opcode for this.

## Examples

Here is an example of the fink opcode. It uses the file *fink.csd* [examples/fink.csd].

### Example 279. Example of the fink opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

Note that this example requires the file *test.txt* to be created, for example by *fouti*

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o fink.wav -W ;; for file output any platform
```

```
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gihand fiopen "test.txt", 1

instr 1

khz init 0
    fink gihand, 0, 1, khz
ar oscil 0.5, khz
    outs ar, ar
endin
</CsInstruments>
<CsScore>

i 1 0 1

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*fin, fini*

## Credits

Author: Gabriel Maldonado  
Italy  
1999

New in Csound version 3.56

# fiopen

`fiopen` — Opens a file in a specific mode.

## Description

*fiopen* can be used to open a file in one of the specified modes.

## Syntax

```
ihandle fiopen ifilename, imode
```

## Initialization

*ihandle* -- a number which specifies this file.

*ifilename* -- the target file's name (in double-quotes).

*imode* -- choose the mode of opening the file. *imode* can be a value chosen among the following:

- 0 - open a text file for writing
- 1 - open a text file for reading
- 2 - open a binary file for writing
- 3 - open a binary file for reading

## Performance

*fiopen* opens a file to be used by the *fout* family of opcodes. It is safer to use it in the header section, external to any instruments. It returns a number, *ihandle*, which unequivocally refers to the opened file.

If *fiopen* is called on an already open file, it just returns the same handle again, and does not close the file.

Notice that *fout* and *foutk* can use either a string containing a file pathname, or a handle-number generated by *fiopen*. Whereas, with *fouti* and *foutir*, the target file can be only specified by means of a handle-number.

## Examples

Here is an example of the `fiopen` opcode. It uses the file *fiopen.csd* [examples/fiopen.csd]. Note that the example must be run from a directory for which the user has write access.

### Example 280. Example of the `fiopen` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
-odac      ;;;realtime audio out
```

```
;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o fiopen.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gihand fiopen "test1.txt", 0

instr 1

ires random 0, 100
fouti gihand, 0, 1, ires
ficlose gihand

endin
</CsInstruments>
<CsScore>

i 1 0 1

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*ficlose fout, fouti, foutir, foutk*

## Credits

Author: Gabriel Maldonado  
Italy  
1999

New in Csound version 3.56

# flanger

flanger — A user controlled flanger.

## Description

A user controlled flanger.

## Syntax

```
ares flanger asig, adel, kfeedback [, imaxd]
```

## Initialization

*imaxd*(optional) -- maximum delay in seconds (needed for initial memory allocation)

## Performance

*asig* -- input signal

*adel* -- delay in seconds

*kfeedback* -- feedback amount (in normal tasks this should not exceed 1, even if bigger values are allowed)

This unit is useful for generating choruses and flangers. The delay must be varied at a-rate, for example by connecting *adel* to an oscillator output. The feedback can vary at k-rate. This opcode is implemented to allow *kr* different than *sr* (else delay could not be lower than *ksmps*) enhancing realtime performance. This unit is very similar to *wguide1*, the only difference is *flanger* does not have the lowpass filter or the requirement that the delay be varied at a-rate.

## Examples

Here is an example of the flanger opcode. It uses the file *flanger.csd* [examples/flanger.csd].

### Example 281. Example of the flanger opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
;-o flanger.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
```

```
instr 1

kfeedback = p4
asnd vco2 .2, 50
adel linseg 0, p3*.5, 0.02, p3*.5, 0 ;max delay time =20ms
aflg flanger asnd, adel, kfeedback
asig clip aflg, 1, 1
      outs asig+asnd, asig+asnd ;mix flanger with original

endin
</CsInstruments>
<CsScore>

i 1 0 10 .2
i 1 11 10 .8 ;lot of feedback
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

More information on flanging on Wikipedia: <http://en.wikipedia.org/wiki/Flanger>

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.49

# flashtxt

flashtxt — Allows text to be displayed from instruments like sliders

## Description

Allows text to be displayed from instruments like sliders etc. (only on Unix and Windows at present)

## Syntax

```
flashtxt iwhich, String
```

## Initialization

*iwhich* -- the number of the window.

*String* -- the string to be displayed.

## Performance

Note that this opcode is not available on Windows due to the implimentation of pipes on that system

A window is created, identified by the *iwhich* argument, with the text string displayed. If the text is replaced by a number then the window id deleted. Note that the text windows are globally numbered so different instruments can change the text, and the window survives the instance of the instrument.

## Examples

Here is an example of the flashtxt opcode. It uses the file *flashtxt.csd* [examples/flashtxt.csd].

### Example 282. Example of the flashtxt opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o flashtxt.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

instr 1
  flashtxt 1, "Instr 1 live"
  ao oscil 4000, 440, 1
```

```
    out ao
  endin

</CsInstruments>
<CsScore>

; Table 1: an ordinary sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for three seconds.
i 1 0 3
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 4.11

# FLbox

FLbox — A FLTK widget that displays text inside of a box.

## Description

A FLTK widget that displays text inside of a box.

## Syntax

```
ihandle FLbox "label", itype, ifont, isize, iwidth, iheight, ix, iy [, image]
```

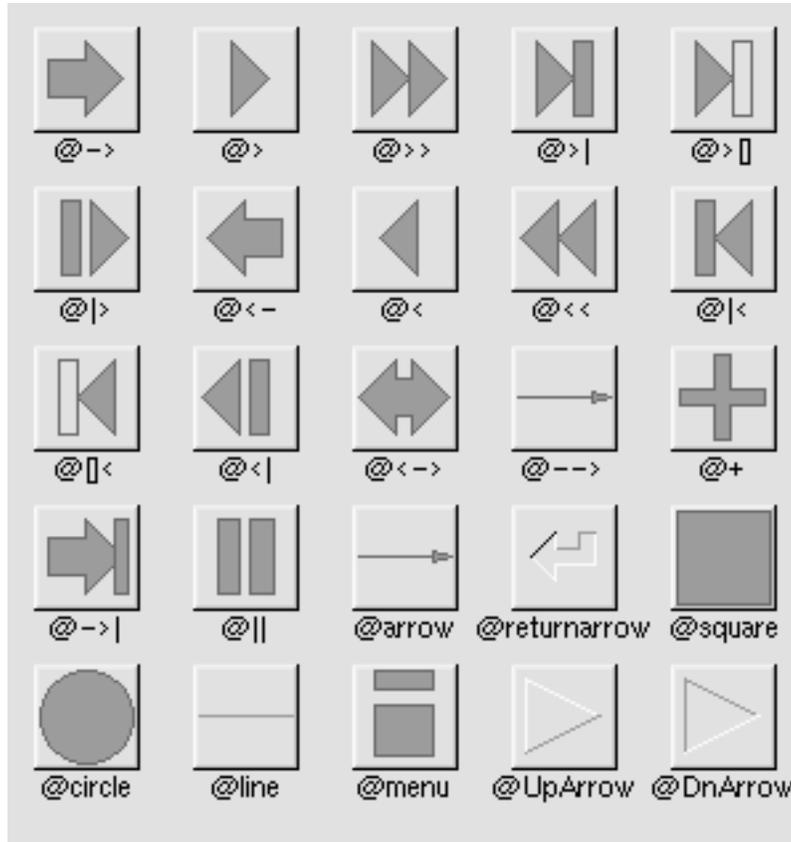
## Initialization

*ihandle* -- a handle value (an integer number) that unequivocally references a corresponding widget. This is used by other opcodes that modify a widget's properties (see *Modifying FLTK Widget Appearance*). It is automatically output by *FLbox* and must not be set by the user label. (The user label is a double-quoted string containing some user-provided text placed near the widget.)

*"label"* -- a double-quoted string containing some user-provided text, placed near corresponding widget.

Notice that with *FLbox*, it is not necessary to call the *FLsetTextType* opcode at all in order to use a symbol. In this case, it is sufficient to set a label starting with "@" followed by the proper formatting string.

The following symbols are supported:



FLTK label supported symbols.

The @ sign may be followed by the following optional “formatting” characters, in this order:

1. “#” forces square scaling rather than distortion to the widget's shape.
2. +[1-9] or -[1-9] tweaks the scaling a little bigger or smaller.
3. [1-9] rotates by a multiple of 45 degrees. “6” does nothing, the others point in the direction of that key on a numeric keypad.

*itype* -- an integer number denoting the appearance of the widget.

The following values are legal for *itype*:

- 1 - flat box
- 2 - up box
- 3 - down box
- 4 - thin up box
- 5 - thin down box
- 6 - engraved box
- 7 - embossed box
- 8 - border box
- 9 - shadow box
- 10 - rounded box
- 11 - rounded box with shadow
- 12 - rounded flat box
- 13 - rounded up box
- 14 - rounded down box
- 15 - diamond up box
- 16 - diamond down box
- 17 - oval box
- 18 - oval shadow box
- 19 - oval flat box

*ifont* -- an integer number denoting the font of *FLbox*.

*ifont* argument to set the font type. The following values are legal for *ifont*:

- 1 - helvetica (same as "Arial" under Windows)
- 2 - helvetica bold
- 3 - helvetica italic
- 4 - helvetica bold italic
- 5 - courier
- 6 - courier bold
- 7 - courier italic
- 8 - courier bold italic
- 9 - times
- 10 - times bold
- 11 - times italic
- 12 - times bold italic
- 13 - symbol
- 14 - screen
- 15 - screen bold
- 16 - dingbats

*isize* -- size of the font.

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* -- horizontal position of the upper left corner of the valuator, relative to the upper left corner of corresponding window. (Expressed in pixels.)

*iy* -- vertical position of the upper left corner of the valuator, relative to the upper left corner of corresponding window. (Expressed in pixels.)

*image* -- a handle referring to an eventual image opened with *bmopen* opcode. If it is set, it allows a skin for that widget.



### **Note about the *bmopen* opcode**

Although the documentation mentions the *bmopen* opcode, it has not been implemented in Csound 4.22.

## **Performance**

*FLbox* is useful to show some text in a window. The text is bounded by a box, whose aspect depends on *itype* argument.

Note that *FLbox* is not a valuator and its value is fixed. Its value cannot be modified.

## Examples

Here is an example of the *FLbox* opcode. It uses the file *FLbox.csd* [examples/FLbox.csd].

### Example 283. Example of the *FLbox* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc      -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o FLbox.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 441
ksmps = 100
nchnls = 1

FLpanel "Text Box", 700, 400, 50, 50
; Box border type (7=embossed box)
itype = 7
; Font type (10='Times Bold')
ifont = 10
; Font size
isize = 20
; Width of the flbox
iwidth = 400
; Height of the flbox
iheight = 30
; Distance of the left edge of the flbox
; from the left edge of the panel
ix = 150
; Distance of the upper edge of the flbox
; from the upper edge of the panel
iy = 100

ih3 FLbox "Use Text Boxes For Labelling", itype, ifont, isize, iwidth, iheight, ix, iy
; End of panel contents
FLpanelEnd
; Run the widget thread!
FLrun

instr 1
endin

</CsInstruments>
<CsScore>

; Real-time performance for 1 hour.
f 0 3600
e

</CsScore>
</CsoundSynthesizer>

```

## See Also

*FLbutBank, FLbutton, FLprintk, FLprintk2, FLvalue*

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.

# FLbutBank

FLbutBank — A FLTK widget opcode that creates a bank of buttons.

## Description

A FLTK widget opcode that creates a bank of buttons.

## Syntax

```
kout, ihandle FLbutBank itype, inumx, inumy, iwidth, iheight, ix, iy, \  
iopcode [, kp1] [, kp2] [, kp3] [, kp4] [, kp5] [... ] [, kpN]
```

## Initialization

*ihandle* -- a handle value (an integer number) that unequivocally references a corresponding widget. This is used by other opcodes that modify a widget's properties (see *Modifying FLTK Widget Appearance*). It is automatically output by *FLbutBank* and must not be set by the user label. (The user label is a double-quoted string containing some user-provided text placed near the widget.)

*itype* -- an integer number denoting the appearance of the widget. The valid numbers are:

- 1 - normal button
- 2 - light button
- 3 - check button
- 4 - round button

You can add 20 to the value to create a "plastic" type button. (Note that there is no Plastic Round button. i.e. if you set type to 24 it will look exactly like type 23).

*inumx* -- number of buttons in each row of the bank.

*inumy* -- number of buttons in each column of the bank

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window, expressed in pixels

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window, expressed in pixels

*iopcode* -- score opcode type. You have to provide the ascii code of the letter corresponding to the score opcode. At present time only "i" (ascii code 105) score statements are supported. A zero value refers to a default value of "i". So both 0 and 105 activates the *i* opcode. A value of -1 disables this opcode feature.

## Performance

*kout* -- output value

*kp1, kp2, ..., kpN* -- arguments of the activated instruments.

The *FLbutBank* opcode creates a bank of buttons. For example, the following line:

```
gkButton,ihbl FLbutBank 22, 8, 8, 380, 180, 50, 350, 0, 7, 0, 0, 5000, 6000
```

will create the this bank:



*FLbutBank*.

A click to a button checks that button. It may also uncheck a previous checked button belonging to the same bank. So the behaviour is always that of radio-buttons. Notice that each button is labeled with a progressive number. The *kout* argument is filled with that number when corresponding button is checked.

*FLbutBank* not only outputs a value but can also activate (or schedule) an instrument provided by the user each time a button is pressed. If the *iopcode* argument is set to a negative number, no instrument is activated so this feature is optional. In order to activate an instrument, *iopcode* must be set to 0 or to 105 (the ascii code of character “i”, referring to the *i* score opcode). P-fields of the activated instrument are *kp1* (instrument number), *kp2* (action time), *kp3* (duration) and so on with user p-fields.

The *itype* argument sets the type of buttons identically to the *FLbutton* opcode. By adding 10 to the *itype* argument (i.e. by setting 11 for type 1, 12 for type 2, 13 for type 3 and 14 for type 4), it is possible to skip the current *FLbutBank* value when getting/setting snapshots (see *General FLTK Widget-related Opcodes*). You can also add 10 to "plastic" button types (31 for type 1, 32 for type 2, etc.)

*FLbutBank* is very useful to retrieve snapshots.

## Examples

Here is an example of the *FLbutBank* opcode. It uses the file *FLbutBank.csd* [examples/*FLbutBank.csd*].

### Example 284. Example of the *FLbutBank* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc      -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o FLbutton.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```
sr = 44100
nchnls = 1

FLpanel "Button Bank", 520, 140, 100, 100
;itype = 2 ;Light Buttons
itype = 22 ;Plastic Light Buttons
inumx = 10
inumy = 4
iwidth = 500
iheight = 120
ix = 10
iy = 10
iopcode = 0
istarttim = 0
idur = 1

gkbutton, ihbb FLbutBank itype, inumx, inumy, iwidth, iheight, ix, iy, iopcode, 1, istarttim, idur

FLpanelEnd
FLrun

instr 1
ibutton = i(gkbutton)
prints "Button %i pushed!\\n", ibutton
endin

</CsInstruments>
<CsScore>

; Real-time performance for 1 hour.
f 0 3600
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*FLbox, FLbutton, FLprintk, FLprintk2, FLvalue*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLbutton

FLbutton — A FLTK widget opcode that creates a button.

## Description

A FLTK widget opcode that creates a button.

## Syntax

```
kout, ihandle FLbutton "label", ion, ioff, itype, iwidth, iheight, ix, \  
iy, iopcode [, kp1] [, kp2] [, kp3] [, kp4] [, kp5] [....] [, kpN]
```

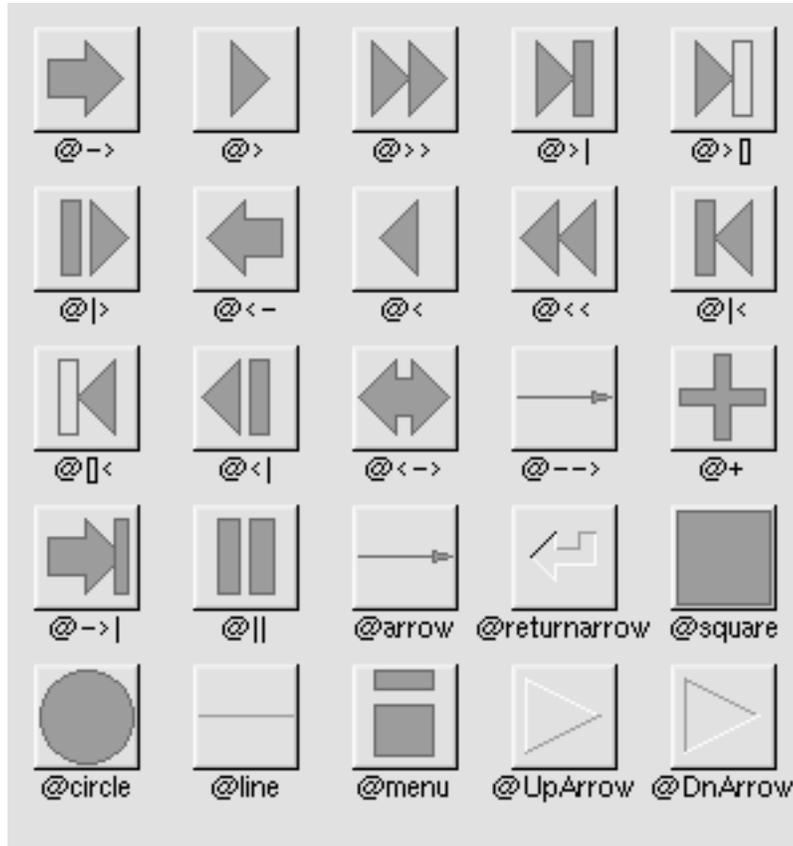
## Initialization

*ihandle* -- a handle value (an integer number) that unequivocally references a corresponding widget. This is used by other opcodes that modify a widget's properties (see *Modifying FLTK Widget Appearance*). It is automatically output by *FLbutton* and must not be set by the user label. (The user label is a double-quoted string containing some user-provided text placed near the widget.)

*“label”* -- a double-quoted string containing some user-provided text, placed near the corresponding widget.

Notice that with *FLbutton*, it is not necessary to call the *FLsetTextType* opcode at all in order to use a symbol. In this case, it is sufficient to set a label starting with “@” followed by the proper formatting string.

The following symbols are supported:



FLTK label supported symbols.

The @ sign may be followed by the following optional “formatting” characters, in this order:

1. “#” forces square scaling rather than distortion to the widget's shape.
2. +[1-9] or -[1-9] tweaks the scaling a little bigger or smaller.
3. [1-9] rotates by a multiple of 45 degrees. “6” does nothing, the others point in the direction of that key on a numeric keypad.

*ion* -- value output when the button is checked.

*ioff* -- value output when the button is unchecked.

*itype* -- an integer number denoting the appearance of the widget.

Several kind of buttons are possible, according to the value of *itype* argument:

- 1 - normal button
- 2 - light button
- 3 - check button
- 4 - round button

You can add 20 to the value to create a "plastic" type button. (Note that there is no Plastic Round button. i.e. if you set type to 24 it will look exactly like type 23).

This is the appearance of the buttons:



FLbutton.

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iopcode* -- score opcode type. You have to provide the ascii code of the letter corresponding to the score opcode. At present time only "i" (ascii code 105) score statements are supported. A zero value refers to a default value of "i". So both 0 and 105 activates the *i* opcode. A value of -1 disables this opcode feature.

## Performance

*kout* -- output value

*kp1, kp2, ..., kpN* -- arguments of the activated instruments.

Buttons of type 2, 3, and 4 also output (*kout* argument) the value contained in the *ion* argument when checked, and that contained in *ioff* argument when unchecked.

By adding 10 to *itype* argument (i.e. by setting 11 for type 1, 12 for type 2, 13 for type 3 and 14 for type 4) it is possible to skip the button value when getting/setting snapshots (see later section). *FLbutton* not only outputs a value, but can also activate (or schedule) an instrument provided by the user each time a button is pressed. You can also add 10 to "plastic" button types (31 for type 1, 32 for type 2, etc.)

If the *iopcode* argument is set to a negative number, no instrument is activated. So this feature is optional. In order to activate an instrument, *iopcode* must be set to 0 or to 105 (the ascii code of character "i", referring to the *i* score opcode).

P-fields of the activated instrument are *kp1* (instrument number), *kp2* (action time), *kp3* (duration) and so on with user p-fields. Notice that in dual state buttons (light button, check button and round button), the instrument is activated only when button state changes from unchecked to checked (not when passing from checked to unchecked).

## Examples

Here is an example of the FLbutton opcode. It uses the file *FLbutton.csd* [examples/FLbutton.csd], and *beats.wav* [examples/beats.wav].

### Example 285. Example of the FLbutton opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc      -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o FLbutton.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Using FLbuttons to create on screen controls for play,
; stop, fast forward and fast rewind of a sound file
; This example also makes use of a preset graphic for buttons.

sr = 44100
kr = 44100
ksmps = 1
nchnls = 2

FLpanel "Buttons", 240, 400, 100, 100
  ion = 0
  ioff = 0
  itype = 1
  iwidth = 50
  iheight = 50
  ix = 10
  iy = 10
  iopcode = 0
  istarttim = 0
  idur = -1 ;Turn instruments on indefinitely

; Normal speed forwards
gkplay, ihb1 FLbutton "@>", ion, ioff, itype, iwidth, iheight, ix, iy, iopcode, 1, istarttim, idur
; Stationary
gkstop, ihb2 FLbutton "@square", ion,ioff, itype, iwidth, iheight, ix+55, iy, iopcode, 2, istarttim, idur
; Double speed backwards
gkrew, ihb3 FLbutton "@<<", ion, ioff, itype, iwidth, iheight, ix + 110, iy, iopcode, 1, istarttim, idur
; Double speed forward
gkff, ihb4 FLbutton "@>>", ion, ioff, itype, iwidth, iheight, ix+165, iy, iopcode, 1, istarttim, idur
; Type 1
gkt1, iht1 FLbutton "1-Normal Button", ion, ioff, 1, 200, 40, ix, iy + 65, -1
; Type 2
gkt2, iht2 FLbutton "2-Light Button", ion, ioff, 2, 200, 40, ix, iy + 110, -1
; Type 3
gkt3, iht3 FLbutton "3-Check Button", ion, ioff, 3, 200, 40, ix, iy + 155, -1
; Type 4
gkt4, iht4 FLbutton "4-Round Button", ion, ioff, 4, 200, 40, ix, iy + 200, -1
; Type 21
```

```
gkt5, iht5 FLbutton "21-Plastic Button", ion, ioff, 21, 200, 40, ix, iy + 245, -1
; Type 22
gkt6, iht6 FLbutton "22-Plastic Light Button", ion, ioff, 22, 200, 40, ix, iy + 290, -1
; Type 23
gkt7, iht7 FLbutton "23-Plastic Check Button", ion, ioff, 23, 200, 40, ix, iy + 335, -1
FLpanelEnd
FLrun

; Ensure that only 1 instance of instr 1
; plays even if the play button is clicked repeatedly
insnum = 1
icount = 1
maxalloc insnum, icount

instr 1
  asig diskin2 "beats.wav", p4, 0, 1
  outs asig, asig
endin

instr 2
  turnoff2 1, 0, 0 ;Turn off instr 1
  turnoff ;Turn off this instrument
endin

</CsInstruments>
<CsScore>

; Real-time performance for 1 hour.
f 0 3600
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*FLbox, FLbutBank, FLprintk, FLprintk2, FLvalue*

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.

# FLcloseButton

FLcloseButton — A FLTK widget opcode that creates a button that will close the panel window it is a part of.

## Description

A FLTK widget opcode that creates a button that will close the panel window it is a part of.

## Syntax

```
ihandle FLcloseButton "label", iwidth, iheight, ix, iy
```

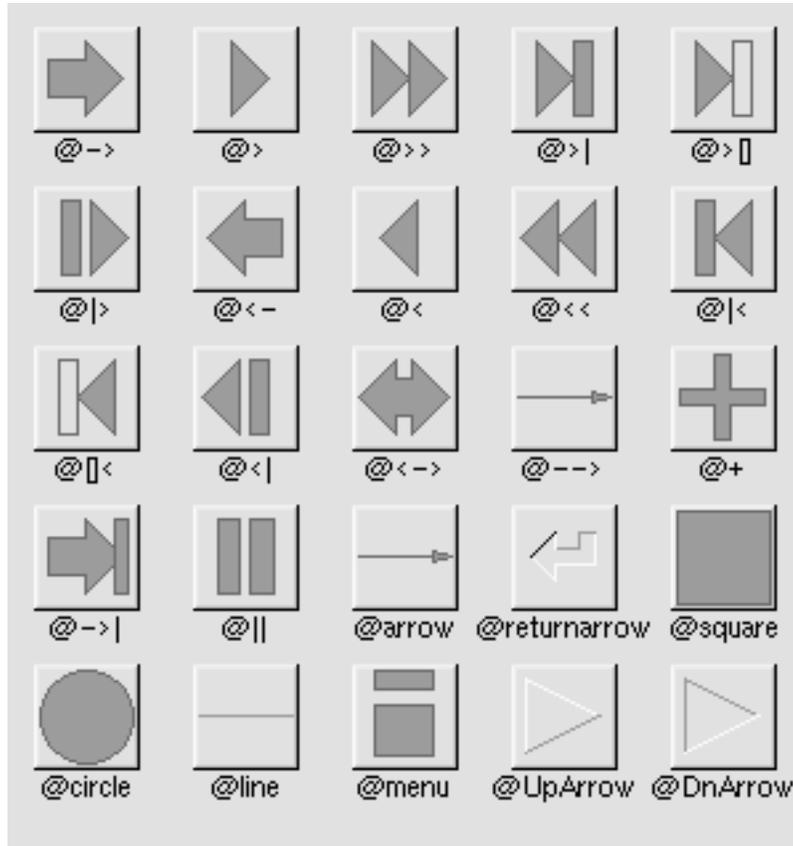
## Initialization

*ihandle* -- a handle value (an integer number) that unequivocally references a corresponding widget. This is used by other opcodes that modify a widget's properties (see *Modifying FLTK Widget Appearance*). It is automatically output by *FLcloseButton* and must not be set by the user label. (The user label is a double-quoted string containing some user-provided text placed near the widget.)

*"label"* -- a double-quoted string containing some user-provided text, placed near the corresponding widget.

Notice that with *FLcloseButton*, it is not necessary to call the *FLsetTextType* opcode at all in order to use a symbol. In this case, it is sufficient to set a label starting with "@" followed by the proper formatting string.

The following symbols are supported:



FLTK label supported symbols.

The @ sign may be followed by the following optional “formatting” characters, in this order:

1. “#” forces square scaling rather than distortion to the widget's shape.
2. +[1-9] or -[1-9] tweaks the scaling a little bigger or smaller.
3. [1-9] rotates by a multiple of 45 degrees. “6” does nothing, the others point in the direction of that key on a numeric keypad.

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

## See Also

*FLbutton*, *FLbox*, *FLbutBank*, *FLprintk*, *FLprintk2*, *FLvalue*

## Credits

Author: Steven Yi

New in version 5.05

# FLcolor

FLcolor — A FLTK opcode that sets the primary colors.

## Description

Sets the primary colors to RGB values given by the user.

## Syntax

```
FLcolor ired, igreen, iblue [, ired2, igreen2, iblue2]
```

## Initialization

*ired* -- The red color of the target widget. The range for each RGB component is 0-255

*igreen* -- The green color of the target widget. The range for each RGB component is 0-255

*iblue* -- The blue color of the target widget. The range for each RGB component is 0-255

*ired2* -- The red component for the secondary color of the target widget. The range for each RGB component is 0-255

*igreen2* -- The green component for the secondary color of the target widget. The range for each RGB component is 0-255

*iblue2* -- The blue component for the secondary color of the target widget. The range for each RGB component is 0-255

## Performance

These opcodes modify the appearance of other widgets. There are two types of such opcodes, those that don't contain the *ihandle* argument which affect all subsequently declared widgets, and those with *ihandle* which affect only a target widget previously defined.

*FLcolor* sets the primary colors to RGB values given by the user. This opcode affects the primary color of (almost) all widgets defined next its location. User can put several instances of *FLcolor* in front of each widget he intend to modify. However, to modify a single widget, it would be better to use the opcode belonging to the second type (i.e. those containing *ihandle* argument).

*FLcolor* is designed to modify the colors of a group of related widgets that assume the same color. The influence of *FLcolor* on subsequent widgets can be turned off by using -1 as the only argument of the opcode. Also, using -2 (or -3) as the only value of *FLcolor* makes all next widget colors randomly selected. The difference is that -2 selects a light random color, while -3 selects a dark random color.

Using *ired2*, *igreen2*, *iblue2* is equivalent to using a separate *FLcolor2*.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLcolor2

FLcolor2 — A FLTK opcode that sets the secondary (selection) color.

## Description

*FLcolor2* is the same of *FLcolor* except it affects the secondary (selection) color.

## Syntax

```
FLcolor2 ired, igreen, iblue
```

## Initialization

*ired* -- The red color of the target widget. The range for each RGB component is 0-255

*igreen* -- The green color of the target widget. The range for each RGB component is 0-255

*iblue* -- The blue color of the target widget. The range for each RGB component is 0-255

## Performance

These opcodes modify the appearance of other widgets. There are two types of such opcodes: those that don't contain the *ihandle* argument which affect all subsequently declared widgets, and those with *ihandle* which affect only a target widget previously defined.

*FLcolor2* is the same of *FLcolor* except it affects the secondary (selection) color. Setting it to -1 turns off the influence of *FLcolor2* on subsequent widgets. A value of -2 (or -3) makes all next widget secondary colors randomly selected. The difference is that -2 selects a light random color, while -3 selects a dark random color.

## See Also

*FLcolor*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLcount

FLcount — A FLTK widget opcode that creates a counter.

## Description

Allows the user to increase/decrease a value with mouse clicks on a corresponding arrow button.

## Syntax

```
kout, ihandle FLcount "label", imin, imax, istep1, istep2, itype, \  
    iwidth, iheight, ix, iy, iopcode [, kp1] [, kp2] [, kp3] [...] [, kpN]
```

## Initialization

*ihandle* -- a handle value (an integer number) that unequivocally references a corresponding widget. Used by further opcodes that changes some valuator's properties. It is automatically set by the corresponding valuator.

*"label"* -- a double-quoted string containing some user-provided text, placed near the corresponding widget.

*imin* -- minimum value of output range

*imax* -- maximum value of output range

*istep1* -- a floating-point number indicating the increment of valuator value corresponding to of each mouse click. *istep1* is for fine adjustments.

*istep2* -- a floating-point number indicating the increment of valuator value corresponding to of each mouse click. *istep2* is for coarse adjustments.

*itype* -- an integer number denoting the appearance of the valuator.

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

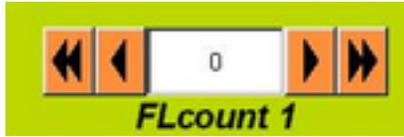
*iopcode* -- score opcode type. You have to provide the ascii code of the letter corresponding to the score opcode. At present time only "i" (ascii code 105) score statements are supported. A zero value refers to a default value of "i". So both 0 and 105 activates the *i* opcode. A value of -1 disables this opcode feature.

## Performance

*kout* -- output value

*kp1, kp2, ..., kpN* -- arguments of the activated instruments.

*FLcount* allows the user to increase/decrease a value with mouse clicks on corresponding arrow buttons:



FLcount.

There are two kind of arrow buttons, for larger and smaller steps. Notice that *FLcount* not only outputs a value and a handle, but can also activate (schedule) an instrument provided by the user each time a button is pressed. P-fields of the activated instrument are *kp1* (instrument number), *kp2* (action time), *kp3* (duration) and so on with user p-fields. If the *iopcode* argument is set to a negative number, no instrument is activated. So this feature is optional.

## Examples

Here is an example of the FLcount opcode. It uses the file *FLcount.csd* [examples/FLcount.csd].

### Example 286. Example of the FLcount opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o FLcount.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Demonstration of the flcount opcode
; clicking on the single arrow buttons
; increments the oscillator in semitone steps
; clicking on the double arrow buttons
; increments the oscillator in octave steps
sr = 44100
kr = 441
ksmps = 100
nchnls = 1

FLpanel "Counter", 900, 400, 50, 50
; Minimum value output by counter
imin = 6
; Maximum value output by counter
imax = 12
; Single arrow step size (semitones)
istep1 = 1/12
; Double arrow step size (octave)
istep2 = 1
; Counter type (1=double arrow counter)
itype = 1
; Width of the counter in pixels
iwidth = 200
; Height of the counter in pixels
iheight = 30
; Distance of the left edge of the counter
; from the left edge of the panel
```

```
ix = 50
; Distance of the top edge of the counter
; from the top edge of the panel
iy = 50
; Score event type (-1=ignored)
iopcode = -1

gkocf, ihandle FLcount "pitch in oct format", imin, imax, istep1, istep2, itype, iwidth, iheight, i
; End of panel contents
FLpanelEnd
; Run the widget thread!
FLrun

instr 1
  iamp = 15000
  ifn = 1
  asig oscili iamp, cpsoct(gkocf), ifn
  out asig
endin

</CsInstruments>
<CsScore>

; Function table that defines a single cycle
; of a sine wave.
f 1 0 1024 10 1

; Instrument 1 will play a note for 1 hour.
i 1 0 3600
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*FLjoy, FLknob, FLroller, FLslider, FLtext*

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.

# FLexecButton

FLexecButton — A FLTK widget opcode that creates a button that executes a command.

## Description

A FLTK widget opcode that creates a button that executes a command. Useful for opening up HTML documentation as About text or to start a separate program from an FLTK widget interface.



### Warning

Because any command can be executed, the user is advised to be very careful when using this opcode and when running orchestras by others using this opcode.

## Syntax

```
ihandle FLexecButton "command", iwidth, iheight, ix, iy
```

## Initialization

*ihandle* -- a handle value (an integer number) that unequivocally references a corresponding widget. This is used by other opcodes that modify a widget's properties (see *Modifying FLTK Widget Appearance*). It is automatically output by *FLexecButton*.

*"command"* -- a double-quoted string containing a command to execute.

Notice that with *FLexecButton*, the default text for the button is "About" and it is necessary to call the *FLsetText* opcode to change the text of the button.

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

## Examples

Here is an example of the FLexecButton opcode. It uses the file *FLexecButton.csd* [examples/FLexecButton.csd].

### Example 287. Example of the FLexecButton opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>

<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No display
-odac          -iadc     -d          ;;RT audio I/O
```

```

</CsOptions>

<CsInstruments>

    sr      = 44100
    ksmps   = 10
    nchnls  = 1

; Example by Jonathan Murphy 2007

;;; reset amplitude range
    odbfs   = 1

;;; set the base colour for the panel
    FLcolor 100, 0, 200
;;; define the panel
    FLpanel "FLexecButton", 250, 100, 0, 0
;;; sliders to control time stretch and pitch
    gkstr, gistretch  FLslider "Time", 0.5, 1.5, 0, 6, -1, 10, 60, 150, 20
    gkpch, gipitch    FLslider "Pitch", 0.5, 1.5, 0, 6, -1, 10, 60, 200, 20
;;; set FLexecButton colour
    FLcolor 255, 255, 0
;;; when this button is pressed, fourier analysis is performed on the file
;;; "beats.wav", producing the analysis file "beats.pvx"
    gipvoc  FLexecButton "csound -U pvanal beats.wav beats.pvx", 60, 20, 20, 20
;;; set FLexecButton text
    FLsetText "PVOC", gipvoc
;;; when this button is pressed, instr 10000 is called, exiting
;;; Csound immediately

;;; cancel previous colour
    FLcolor -1
;;; set colour for kill button
    FLcolor 255, 0, 0
    gkkill, gikill  FLbutton "X", 1, 1, 1, 20, 20, 100, 20, 0, 10000, 0, 0.1
;;; cancel previous colour
    FLcolor -1
;;; set colour for play/stop and pause buttons
    FLcolor 0, 200, 0
;;; pause and play/stop buttons
    gkpause, gipause  FLbutton "@| |", 1, 0, 2, 40, 20, 20, 60, -1
    gkplay, giplay    FLbutton "@|>", 1, 0, 2, 40, 20, 80, 60, -1
;;; end the panel
    FLpanelEnd
;;; set initial values for time stretch and pitch
    FLsetVal_i 1, gistretch
    FLsetVal_i 1, gipitch
;;; run the panel
    FLrun

    instr 1          ; trigger play/stop
    ;; is the play/stop button on or off?
    ;; either way we need to trigger something,
    ;; so we can't just use the value of gkplay
    kon  trigger  gkplay, 0, 0
    koff trigger  gkplay, 1, 1
    ;; if on, start instr 2
    schedkwhen kon, -1, -1, 2, 0, -1
    ;; if off, stop instr 2
    schedkwhen koff, -1, -1, -2, 0, -1

    endin

    instr 2

    ;; paused or playing?

```

```

if (gkpause == 1) kgoto pause
    kgoto    start
pause:
;; if the pause button is on, skip sound production
    kgoto    end
start:
;; get the length of the analysis file in seconds
ilen    filelen    "beats.pvx"
;; determine base frequency of playback
icps    = 1/ilen
;; create a table over the length of the file
itpt    ftgen      0, 0, 513, -7, 0, 512, ilen
;; phasor for time control
kphs    phasor     icps * gkstr
;; use phasor as index into table
kndx    = kphs * 512
;; read table
ktpt    tablei     kndx, itpt
;; use value from table as time pointer into file
fsig1    pvsfread  ktpt, "beats.pvx"
;; change playback pitch
fsig2    pvscale   fsig1, gkpch
;; resynthesize
aout     pvsynth   fsig2
;; envelope to avoid clicks and clipping
aenv     linsegr   0, 0.3, 0.75, 0.1, 0
aout     = aout * aenv
    out        aout
end:

    endin

instr 10000      ; kill

    exitnow

    endin

</CsInstruments>

<CsScore>
i1 0 10000
e
</CsScore>

</CsoundSynthesizer>

```

## See Also

*FLbutton, FLbox, FLbutBank, FLprintk, FLprintk2, FLvalue*

## Credits

Author: Steven Yi

Example by: Jonathan Murphy

New in version 5.05

# FLgetsnap

FLgetsnap — Retrieves a previously stored FLTK snapshot.

## Description

Retrieves a previously stored snapshot (in memory), i.e. sets all valuator to the corresponding values stored in that snapshot.

## Syntax

```
inumsnap FLgetsnap index [, igroup]
```

## Initialization

*inumsnap* -- current number of snapshots.

*index* -- a number referring unequivocally to a snapshot. Several snapshots can be stored in the same bank.

*igroup* -- (optional) an integer number referring to a snapshot-related group of widget. It allows to get/set, or to load/save the state of a subset of valuators. Default value is zero that refers to the first group. The group number is determined by the opcode *FLsetSnapGroup*.



### Note

The *igroup* parameter has not been yet fully implemented in the current version of csound. Please do not rely on it yet.

## Performance

*FLgetsnap* retrieves a previously stored snapshot (in memory), i.e. sets all valuator to the corresponding values stored in that snapshot. The *index* argument unequivocally must refer to an already existing snapshot. If the *index* argument refers to an empty snapshot or to a snapshot that doesn't exist, no action is done. *FLsetsnap* outputs the current number of snapshots (*inumsnap* argument).

For purposes of snapshot saving, widgets can be grouped, so that snapshots affect only a defined group of widgets. The opcode *FLsetSnapGroup* is used to specify the group for all widgets declared after it, until the next *FLsetSnapGroup* statement.

## See Also

*FLloadsnap*, *FLrun*, *FLsavesnap*, *FLsetsnap*, *FLsetSnapGroup*, *FLupdate*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLgroup

FLgroup — A FLTK container opcode that groups child widgets.

## Description

A FLTK container opcode that groups child widgets.

## Syntax

```
FLgroup "label", iwidth, iheight, ix, iy [, iborder] [, image]
```

## Initialization

*“label”* -- a double-quoted string containing some user-provided text, placed near the corresponding widget.

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iborder* (optional, default=0) -- border type of the container. It is expressed by means of an integer number chosen from the following:

- 0 - no border
- 1 - down box border
- 2 - up box border
- 3 - engraved border
- 4 - embossed border
- 5 - black line border
- 6 - thin down border
- 7 - thin up border

If the integer number doesn't match any of the previous values, no border is provided as the default.

*image* (optional) -- a handle referring to an eventual image opened with the *bmopen* opcode. If it is set, it allows a skin for that widget.



### Note about the *bmopen* opcode

Although the documentation mentions the *bmopen* opcode, it has not been implemented in Csound 4.22.

## Performance

Containers are useful to format the graphic appearance of the widgets. The most important container is *FLpanel*, that actually creates a window. It can be filled with other containers and/or valuators or other kinds of widgets.

There are no k-rate arguments in containers.

## See Also

*FLgroupEnd, FLpack, FLpackEnd, FLpanel, FLpanelEnd, FLscroll, FLscrollEnd, FLtabs, FLtabsEnd*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLgroupEnd

FLgroupEnd — Marks the end of a group of FLTK child widgets.

## Description

Marks the end of a group of FLTK child widgets.

## Syntax

```
FLgroupEnd
```

## Performance

Containers are useful to format the graphic appearance of the widgets. The most important container is *FLpanel*, that actually creates a window. It can be filled with other containers and/or valuators or other kinds of widgets.

There are no k-rate arguments in containers.

## See Also

*FLgroup*, *FLpack*, *FLpackEnd*, *FLpanel*, *FLpanelEnd*, *FLscroll*, *FLscrollEnd*, *FLtabs*, *FLtabsEnd*

## Credits

Author: Gabriel Maldonado

New in version 4.22

## FLgroup\_end

FLgroup\_end — Marks the end of a group of FLTK child widgets.

### Description

Marks the end of a group of FLTK child widgets. This is another name for **FLgroupEnd** provides for compatibility. See *FLgroupEnd*

### Credits

Author: Gabriel Maldonado

New in version 4.22

# FLhide

FLhide — Hides the target FLTK widget.

## Description

Hides the target FLTK widget, making it invisible.

## Syntax

```
FLhide ihandle
```

## Initialization

*ihandle* -- a handle value (an integer number) that unequivocally references a corresponding widget. This is used by other opcodes that modify a widget's properties (see *Modifying FLTK Widget Appearance*).

## Performance

*FLhide* hides target widget, making it invisible.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLhvsBox

FLhvsBox — Displays a box with a grid useful for visualizing two-dimensional Hyper Vectorial Synthesis.

## Description

*FLhvsBox* displays a box with a grid useful for visualizing two-dimensional Hyper Vectorial Synthesis.

## Syntax

```
ihandle FLhvsBox inumlinesX, inumlinesY, iwidth, iheight, ix, iy
```

## Initialization

*ihandle* – an integer number used a univocally-defined handle for identifying a specific HVS box (see below).

*inumlinesX*, *inumlinesY* - number of vertical and horizontal lines delimiting the HVS squared areas

*iwidth*, *iheight* - width and height of the HVS box

*ix*, *iy* - the position of the HVS box

## Performance

*FLhvsBox* is a widget able to visualize current position of the HVS cursor in an HVS box (i.e. a squared area containing a grid). The number of horizontal and vertical lines of the grid can be defined with the *inumlinesX*, *inumlinesY* arguments. This opcode has to be declared inside an *FLpanel - FLpanelEnd* block. See the entry for *hvs2* for an example of usage of *FLhvsBox*.

*FLhvsBoxSetValue* is used to set the cursor position of an *FLhvsBox* widget.

## See Also

*hvs2*, *FLhvsBoxSetValue*

## Credits

Author: Gabriel Maldonado

New in version 5.06

# FLhvsBoxSetValue

FLhvsBoxSetValue — Sets the cursor position of a previously-declared FLhvsBox widget.

## Description

*FLhvsBoxSetValue* sets the cursor position of a previously-declared *FLhvsBox* widget.

## Syntax

```
FLhvsBox kx, ky, ihandle
```

## Initialization

*ihandle* – an integer number used a univocally-defined handle for identifying a specific HVS box (see below).

## Performance

*kx*, *ky*– the coordinates of the HVS cursor position to be set.

*FLhvsBoxSetValue* sets the cursor position of a previously-declared *FLhvsBox* widget. The *kx* and *ky* arguments, denoting the cursor position, have to be expressed in normalized values (0 to 1 range).

See the entry for *hvs2* for an example of usage of *FLhvsBoxSetValue*.

## See Also

*hvs2*, *FLhvsBox*

## Credits

Author: Gabriel Maldonado

New in version 5.06

# FLjoy

FLjoy — A FLTK opcode that acts like a joystick.

## Description

*FLjoy* is a squared area that allows the user to modify two output values at the same time. It acts like a joystick.

## Syntax

```
koutx, kouty, ihandlex, ihandley FLjoy "label", iminx, imaxx, iminy, \  
imaxy, iexpx, iexpy, idispx, idispy, iwidth, iheight, ix, iy
```

## Initialization

*ihandlex* -- a handle value (an integer number) that unequivocally references a corresponding widget. Used by further opcodes that changes some valuator's properties. It is automatically set by the corresponding valuator.

*ihandley* -- a handle value (an integer number) that unequivocally references a corresponding widget. Used by further opcodes that changes some valuator's properties. It is automatically set by the corresponding valuator.

*"label"* -- a double-quoted string containing some user-provided text, placed near the corresponding widget.

*iminx* -- minimum x value of output range

*imaxx* -- maximum x value of output range

*iminy* -- minimum y value of output range

*imaxy* -- maximum y value of output range

*iwidth* -- width of widget.

*idispx* -- a handle value that was output from a previous instance of the *FLvalue* opcode to display the current value of the current valuator in the *FLvalue* widget itself. If the user doesn't want to use this feature that displays current values, it must be set to a negative number by the user.

*idispy* -- a handle value that was output from a previous instance of the *FLvalue* opcode to display the current value of the current valuator in the *FLvalue* widget itself. If the user doesn't want to use this feature that displays current values, it must be set to a negative number by the user.

*iexpx* -- an integer number denoting the behaviour of valuator:

- 0 = valuator output is linear
- -1 = valuator output is exponential

All other positive numbers for *iexpx* indicate the number of an existing table that is used for indexing. Linear interpolation is provided in table indexing. A negative table number suppresses interpolation.

*ixpy* -- an integer number denoting the behaviour of valuator:

- 0 = valuator output is linear
- -1 = valuator output is exponential

All other positive numbers for *ixpy* indicate the number of an existing table that is used for indexing. Linear interpolation is provided in table indexing. A negative table number suppresses interpolation.



## IMPORTANT!

Notice that the tables used by valuator must be created with the *ftgen* opcode and placed in the orchestra before the corresponding valuator. They can not placed in the score. In fact, tables placed in the score are created later than the initialization of the opcodes placed in the header section of the orchestra.

*iheight* -- height of widget.

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

## Performance

*koutx* -- x output value

*kouty* -- y output value

## Examples

Here is an example of the FLjoy opcode. It uses the file *FLjoy.csd* [examples/FLjoy.csd].

### Example 288. Example of the FLjoy opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc    -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o FLjoy.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Demonstration of the flpanel opcode
; Horizontal click-dragging controls the frequency of the oscillator
; Vertical click-dragging controls the amplitude of the oscillator
sr = 44100
kr = 441
ksmps = 100
nchnls = 1

FLpanel "X Y Panel", 900, 400, 50, 50
```

```

; Minimum value output by x movement (frequency)
iminx = 200
; Maximum value output by x movement (frequency)
imaxx = 5000
; Minimum value output by y movement (amplitude)
iminy = 0
; Maximum value output by y movement (amplitude)
imaxy = 15000
; Logarithmic change in x direction
iexpx = -1
; Linear change in y direction
iexpy = 0
; Display handle x direction (-1=not used)
idispx = -1
; Display handle y direction (-1=not used)
idispy = -1
; Width of the x y panel in pixels
iwidth = 800
; Height of the x y panel in pixels
iheight = 300
; Distance of the left edge of the x y panel from
; the left edge of the panel
ix = 50
; Distance of the top edge of the x y
; panel from the top edge of the panel
iy = 50

gkfreqx, gkampy, ihandlex, ihandley FLjoy "X - Frequency Y - Amplitude", iminx, imaxx, iminy, imaxy
; End of panel contents
FLpanelEnd
; Run the widget thread!
FLrun

instr 1
  ifn = 1
  asig oscili gkampy, gkfreqx, ifn
  out asig
endin

</CsInstruments>
<CsScore>

; Function table that defines a single cycle
; of a sine wave.
f 1 0 1024 10 1

; Instrument 1 will play a note for 1 hour.
i 1 0 3600
e

</CsScore>
</CsSoundSynthesizer>

```

## See Also

*FLcount, FLknob, FLroller, FLslider, FLtext*

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.

# FLkeyIn

FLkeyIn — Reports keys pressed (on alphanumeric keyboard) when an FLTK panel has focus.

## Description

*FLkeyIn* informs about the status of a key pressed by the user on the alphanumeric keyboard when an FLTK panel has got the focus.

## Syntax

```
kascii FLkeyIn [ifn]
```

## Initialization

ifn – (optional, default value is zero) set the behavior of FLkeyIn (see below).

## Performance

*kascii* - the ascii value of last pressed key. If the key is pressed, the value is positive, when the key is released the value is negative.

*FLkeyIn* is useful to know whether a key has been pressed on the computer keyboard. The behavior of this opcode depends on the optional *ifn* argument.

If *ifn* = 0 (default), *FLkeyIn* outputs the ascii code of the last pressed key. If it is a special key (ctrl, shift, alt, f1-f12 etc.), a value of 256 is added to the output value in order to distinguish it from normal keys. The output will continue to output the last key value, until a new key is pressed or released. Notice that the output will be negative when a key is depressed.

If *ifn* is set to the number of an already-allocated table having at least 512 elements, then the table element having index equal to the ascii code of the key pressed is set to 1, all other table elements are set to 0. This allows to check the state of a certain key or set of keys.

Be aware that you must set the *ikbdcapture* parameter to something other than 0 on a designated *FLpanel* for *FLkeyIn* to capture keyboard events from that panel.



### Note

*FLkeyIn* works internally at k-rate, so it can't be used in the header as other FLTK opcodes. It must be used inside an instrument.

## Examples

Here is an example of the FLkeyIn opcode. It uses the file *FLkeyIn.csd* [examples/FLkeyIn.csd].

### Example 289. Example of the FLkeyIn opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      ;;RT audio I/O
</CsOptions>
<CsInstruments>

sr=44100
ksmps=128
nchnls=2

;Example by Andres Cabrera 2007

FLpanel "FLkeyIn", 400, 300, -1, -1, 5, 1, 1
FLpanelEnd

FLrun

Odbfs = 1

instr 1
kascii  FLkeyIn
ktrig  changed kascii
if (kascii > 0) then
  printf "Key Down: %i\n", ktrig, kascii
else
  printf "Key Up: %i\n", ktrig, -kascii
endif
endin

</CsInstruments>
<CsScore>
i 1 0 120
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Gabriel Maldonado

New in version 5.06

# FLknob

FLknob — A FLTK widget opcode that creates a knob.

## Description

A FLTK widget opcode that creates a knob.

## Syntax

```
kout, ihandle FLknob "label", imin, imax, iexp, itype, idisp, iwidth, \  
ix, iy [, icursorsize]
```

## Initialization

*ihandle* -- a handle value (an integer number) that unequivocally references a corresponding widget. This is used by other opcodes that modify a widget's properties (see *Modifying FLTK Widget Appearance*). It is automatically output by *FLknob* and must not be set by the user label. (The user label is a double-quoted string containing some user-provided text placed near the widget.)

*label* -- a double-quoted string containing some user-provided text, placed near the corresponding widget.

*imin* -- minimum value of output range.

*imax* -- maximum value of output range.

*iexp* -- an integer number denoting the behaviour of valuator:

- 0 = valuator output is linear
- -1 = valuator output is exponential

All other positive numbers for *iexp* indicate the number of an existing table that is used for indexing. Linear interpolation is provided in table indexing. A negative table number suppresses interpolation.



### IMPORTANT!

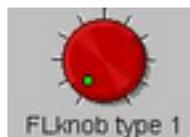
Notice that the tables used by valuator must be created with the *ftgen* opcode and placed in the orchestra before the corresponding valuator. They can not be placed in the score. In fact, tables placed in the score are created later than the initialization of the opcodes placed in the header section of the orchestra.

*itype* -- an integer number denoting the appearance of the valuator.

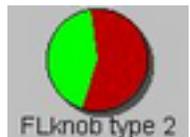
The *itype* argument can be set to the following values:

- 1 - a 3-D knob
- 2 - a pie-like knob
- 3 - a clock-like knob

- 4 - a flat knob



A 3-D knob.



A pie knob.



A clock knob.



A flat knob.

*idisp* -- a handle value that was output from a previous instance of the *FLvalue* opcode to display the current value of the current valuator in the *FLvalue* widget itself. If the user doesn't want to use this feature that displays current values, it must be set to a negative number by the user.

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*icursorsize* (optional) -- If *FLknob's* *itype* is set to 1 (3D knob), this parameter controls the size of knob cursor.

## Performance

*kout* -- output value

*FLknob* puts a knob in the corresponding container.

## Examples

Here is an example of the *FLknob* opcode. It uses the file *FLknob.csd* [examples/FLknob.csd].

**Example 290. Example of the FLknob opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc      -d          ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o FLknob.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; A sine with oscillator with flknob controlled frequency
sr = 44100
kr = 441
ksmps = 100
nchnls = 1

FLpanel "Frequency Knob", 900, 400, 50, 50
; Minimum value output by the knob
imin = 200
; Maximum value output by the knob
imax = 5000
; Logarithmic type knob selected
iexp = -1
; Knob graphic type (1=3D knob)
itype = 1
; Display handle (-1=not used)
idisp = -1
; Width of the knob in pixels
iwidth = 70
; Distance of the left edge of the knob
; from the left edge of the panel
ix = 70
; Distance of the top edge of the knob
; from the top of the panel
iy = 125

gkfreq, ihandle FLknob "Frequency", imin, imax, iexp, itype, idisp, iwidth, ix, iy
; End of panel contents
FLpanelEnd
; Run the widget thread!
FLrun

; Set the widget's initial value
FLsetVal_i 300, ihandle

instr 1
iamp = 15000
ifn = 1
asig oscili iamp, gkfreq, ifn
out asig
endin

</CsInstruments>
<CsScore>

; Function table that defines a single cycle
; of a sine wave.
f 1 0 1024 10 1

; Instrument 1 will play a note for 1 hour.

```

```
i 1 0 3600
e
```

```
</CsScore>
</CsoundSynthesizer>
```

Here is another example of the FLknob opcode, showing the different styles of knobs and the usage of FLvalue to display a knob's value. It uses the file *FLknob-2.csd* [examples/FLknob-2.csd].

### Example 291. More complex example of the FLknob opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc      -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o FLknob.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 441
ksmps = 100
nchnls = 1

;By Andres Cabrera 2007
FLpanel "Knob Types", 330, 230, 50, 50
; Distance of the left edge of the knob
; from the left edge of the panel
ix = 20
; Distance of the top edge of the knob
; from the top of the panel
iy = 20

;Create boxes that display a widget's value
ihandleA FLvalue "A", 60, 20, ix + 130, iy + 110
ihandleB FLvalue "B", 60, 20, ix + 220, iy + 110
ihandleC FLvalue "C", 60, 20, ix + 130, iy + 160
ihandleD FLvalue "D", 60, 20, ix + 220, iy + 160

; The four types of FLknobs
gkdummy1, ihandle1 FLknob "Type 1", 200, 5000, -1, 1, ihandleA, 70, ix, iy, 90
gkdummy2, ihandle2 FLknob "Type 2", 200, 5000, -1, 2, ihandleB, 70, ix + 100, iy
gkdummy3, ihandle3 FLknob "Type 3", 200, 5000, -1, 3, ihandleC, 70, ix + 200, iy
gkdummy4, ihandle4 FLknob "Type 4", 200, 5000, -1, 4, ihandleD, 70, ix, iy + 100
; End of panel contents
FLpanelEnd
; Run the widget thread!
FLrun

; Set the color of widgets
FLsetColor 20, 23, 100, ihandle1
FLsetColor 0, 123, 100, ihandle2
FLsetColor 180, 23, 12, ihandle3
FLsetColor 10, 230, 0, ihandle4

FLsetColor2 200, 230, 0, ihandle1
FLsetColor2 200,0,123, ihandle2
FLsetColor2 180, 180, 100, ihandle3
FLsetColor2 180, 23, 12, ihandle4

; Set the initial value of the widget
```

```
FLsetVal_i 300, ihandle1
FLsetVal_i 1000, ihandle2

instr 1
; Nothing here for now
endin

</CsInstruments>
<CsScore>

f 0 3600 ;Dumy table to make csound wait for realtime events

e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*FLcount, FLjoy, FLroller, FLslider, FLtext*

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.

# FLlabel

FLlabel — A FLTK opcode that modifies the appearance of a text label.

## Description

Modifies a set of parameters related to the text label appearance of a widget (i.e. size, font, alignment and color of corresponding text).

## Syntax

```
FLlabel isize, ifont, ialign, ired, igreen, iblue
```

## Initialization

*isize* -- size of the font of the target widget. Normal values are in the order of 15. Greater numbers enlarge font size, while smaller numbers reduce it.

*ifont* -- sets the the font type of the label of a widget.

Legal values for ifont argument are:

- 1 - Helvetica (same as Arial under Windows)
- 2 - Helvetica Bold
- 3 - Helvetica Italic
- 4 - Helvetica Bold Italic
- 5 - Courier
- 6 - Courier Bold
- 7 - Courier Italic
- 8 - Courier Bold Italic
- 9 - Times
- 10 - Times Bold
- 11 - Times Italic
- 12 - Times Bold Italic
- 13 - Symbol
- 14 - Screen
- 15 - Screen Bold
- 16 - Dingbats

*ialign* -- sets the alignment of the label text of the widget.

Legal values for *ialign* argument are:

- 1 - align center
- 2 - align top
- 3 - align bottom
- 4 - align left
- 5 - align right
- 6 - align top-left
- 7 - align top-right
- 8 - align bottom-left
- 9 - align bottom-right

*ired* -- The red color of the target widget. The range for each RGB component is 0-255

*igreen* -- The green color of the target widget. The range for each RGB component is 0-255

*iblue* -- The blue color of the target widget. The range for each RGB component is 0-255

## Performance

*FLlabel* modifies a set of parameters related to the text label appearance of a widget, i.e. size, font, alignment and color of corresponding text. This opcode affects (almost) all widgets defined next its location. A user can put several instances of *FLlabel* in front of each widget he intends to modify. However, to modify a particular widget, it is better to use the opcode belonging to the second type (i.e. those containing the *ihandle* argument).

The influence of *FLlabel* on the next widget can be turned off by using -1 as its only argument. *FLlabel* is designed to modify text attributes of a group of related widgets.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLloadsnap

FLloadsnap — Loads all snapshots into the memory bank of the current orchestra.

## Description

*FLloadsnap* loads all the snapshots contained in a file into the memory bank of the current orchestra.

## Syntax

```
FLloadsnap "filename" [, igroup]
```

## Initialization

*"filename"* -- a double-quoted string corresponding to a file to load a bank of snapshots.

*igroup* -- (optional) an integer number referring to a snapshot-related group of widget. It allows to get/set, or to load/save the state of a subset of valuator. Default value is zero that refers to the first group. The group number is determined by the opcode *FLsetSnapGroup*.



### Note

The *igroup* parameter has not been yet fully implemented in the current version of csound. Please do not rely on it yet.

## Performance

*FLloadsnap* loads all snapshots contained in filename into the memory bank of current orchestra.

For purposes of snapshot saving, widgets can be grouped, so that snapshots affect only a defined group of widgets. The opcode *FLsetSnapGroup* is used to specify the group for all widgets declared after it, until the next *FLsetSnapGroup* statement.

## See Also

*FLgetsnap*, *FLrun*, *FLsetSnapGroup*, *FLsavesnap*, *FLsetsnap*, *FLupdate*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLmouse

FLmouse — Returns the mouse position and the state of the three mouse buttons.

## Description

*FLmouse* returns the coordinates of the mouse position within an FLTK panel and the state of the three mouse buttons.

## Syntax

```
kx, ky, kb1, kb2, kb3 FLmouse [imode]
```

## Initialization

*imode* – (optional, default = 0) Determines the mode for mouse location reporting.

- 0 - Absolute position normalized to range 0-1
- 1 - Absolute raw pixel position
- 2 - Raw pixel position, relative to FLTK panel

## Performance

*kx, ky* – the mouse coordinates, whose range depends on the *imode* argument (see above).

*kb1, kb2, kb3* – the states of the mouse buttons, 1 when corresponding button is pressed, 0 when the button is not pressed.

*FLmouse* returns the coordinates of the mouse position and the state of the three mouse buttons. The coordinates can be retrieved in three modes depending on the *imode* argument value (see above). Modes 0 and 1 report mouse position in relation to the complete screen (Absolute mode), while mode 2, reports the pixel position within an FLTK panel. Notice that FLmouse is only active when the mouse cursor passes on an *FLpanel* area.

## Examples

Here is an example of the FLmouse opcode. It uses the file *FLmouse.csd* [examples/FLmouse.csd].

### Example 292. Example of the FLmouse opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      ;;RT audio I/O
</CsOptions>
<CsInstruments>
```

```
sr=44100
ksmps=128
nchnls=2

;Example by Andres Cabrera 2007
giwidth = 400
giheight = 300
FLpanel "FLmouse", giwidth, giheight, 10, 10
FLpanelEnd

FLrun

Odbfs = 1

instr 1
  kx, ky, kb1, kb2, kb3   FLmouse 2
  ktrig changed kx, ky ;Print only if coordinates have changed
  printf "kx = %f   ky = %f \n", ktrig, kx, ky
  kfreq = ((giwidth - ky)*1000/giwidth) + 300

  ; y coordinate determines frequency, x coordinate determines amplitude
  ; Left mouse button (kb1) doubles the frequency
  ; Right mouse button (kb3) activates sound on channel 2
  aout oscil kx /giwidth , kfreq * (kb1 + 1), 1
  outs aout, aout * kb3
endin

</CsInstruments>
<CsScore>
f 1 0 1024 10 1

i 1 0 120
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Gabriel Maldonado

New in version 5.06

# flooper

flooper — Function-table-based crossfading looper.

## Description

This opcode reads audio from a function table and plays it back in a loop with user-defined start time, duration and crossfade time. It also allows the pitch of the loop to be controlled, including reversed playback. It accepts non-power-of-two tables, such as deferred-allocation GEN01 tables, with one or two channels.

## Syntax

```
asig1[, asig2] flooper kamp, kpitch, istart, idur, ifad, ifn
```

## Initialization

*istart* -- loop start pos in seconds

*idur* -- loop duration in seconds

*ifad* -- crossfade duration in seconds

*ifn* -- function table number, generally created using GEN01

## Performance

*asig[,asig2]* -- output sig (mono or stereo)

*kamp* -- amplitude control

*kpitch* -- pitch control (transposition ratio); negative values play the loop back in reverse

## Examples

### Example 293.

```
aout flooper 16000, 1, 1, 4, 0.05, 1 ; loop starts at 1 sec, for 4 secs, 0.05 crossfade  
out      aout
```

The example above shows the basic operation of *flooper*. Pitch can be controlled at the k-rate, as well as amplitude. The example assumes table 1 to contain at least 5.05 seconds of audio (4 secs loop duration, starting 1 sec into the table, using 0.05 secs after the loop end for the crossfade).

Here is another example of the flooper opcode. It uses the file *flooper.csd* [examples/flooper.csd] and *fox.wav* [examples/fox.wav].

### Example 294.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o flooper.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kpitch line 1, p3, .9 ;lower pitch a bit during the note
aout flooper .9, kpitch, 1, .53, 0.05, 1 ; loop starts at 1 sec, for .53 secs, 0.05 crossfade
outs aout, aout

endin
</CsInstruments>
<CsScore>
;table size is deferred,
; and format taken from the soundfile header
f 1 0 0 1 "fox.wav" 0 0 0

i 1 0 8.2
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
April 2005

New plugin in version 5

April 2005.

# flooper2

flooper2 — Function-table-based crossfading looper.

## Description

This opcode implements a crossfading looper with variable loop parameters and three looping modes, optionally using a table for its crossfade shape. It accepts non-power-of-two tables for its source sounds, such as deferred-allocation GEN01 tables, with one or two channels.

## Syntax

```
asig1[,asig2] flooper2 kamp, kpitch, kloopstart, kloopend, kcrossfade, ifn \  
[, istart, imode, ifenv, iskip]
```

## Initialization

*ifn* -- sound source function table number, generally created using GEN01

*istart* -- playback start pos in seconds

*imode* -- loop modes: 0 forward, 1 backward, 2 back-and-forth [def: 0]

*ifenv* -- if non-zero, crossfade envelope shape table number. The default, 0, sets the crossfade to linear.

*iskip* -- if 1, the opcode initialisation is skipped, for tied notes, performance continues from the position in the loop where the previous note stopped. The default, 0, does not skip initialisation

## Performance

*asig1*[, *asig2*] -- output sig (mono or stereo).

*kamp* -- amplitude control

*kpitch* -- pitch control (transposition ratio); negative values are not allowed.

*kloopstart* -- loop start point (secs). Note that although k-rate, loop parameters such as this are only updated once per loop cycle.

*kloopend* -- loop end point (secs), updated once per loop cycle.

*kcrossfade* -- crossfade length (secs), updated once per loop cycle and limited to loop length.

Mode 1 for *imode* will only loop backwards from the end point to the start point.

## Examples

### Example 295.

```
aout flooper2 16000, 1, 1, 5, 0.05, 1 ; loop starts at 1 sec, for 4 secs, 0.05 crossfade  
out      aout
```

The example above shows the basic operation of *flooper2*. Pitch can be controlled at the k-rate, as well as amplitude and loop parameters. The example assumes table 1 to contain at least 5.05 seconds of audio (4 secs loop duration, starting 1 sec into the table, using 0.05 secs after the loop end for the crossfade). Looping is in mode 0 (normal forward loop).

Here is another example of the *flooper2* opcode. It uses the file *flooper2.csd* [examples/flooper2.csd] and *fox.wav* [examples/fox.wav].

### Example 296.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
-iadc      ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o flooper2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
; looping back and forth, 0.05 crossfade
kst line .2, p3, 2 ;vary loopstartpoint
aout flooper2 .8, 1, 0, kst, 0.05, 1, 0, 2
outs aout, aout

endin
</CsInstruments>
<CsScore>
; Its table size is deferred,
; and format taken from the soundfile header
f 1 0 0 1 "fox.wav" 0 0 0

i 1 0 12
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
July 2006

New plugin in version 5

July 2006.

# floor

floor — Returns the largest integer not greater than  $x$

## Description

Returns the largest integer not greater than  $x$

## Syntax

`floor(x)` (init-, control-, or audio-rate arg allowed)

`floor(k/i[])` (k- or i-arrays )

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

## Examples

Here is an example of the floor opcode. It uses the file *floor.csd* [examples/floor.csd].

### Example 297. Example of the floor opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
;-o floor.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

idiv init 1

loop:
inumber = 9
i1 = inumber / idiv
ifl = floor(i1)
print inumber, idiv, ifl ;print number / idiv = result using floor
idiv = idiv + 1
if (idiv <= 10) igoto loop

endin
</CsInstruments>
<CsScore>
```

```
i 1 0 0  
e  
  
</CsScore>  
</CsoundSynthesizer>
```

The output should include lines like these:

```
instr 1: inumber = 9.000 idiv = 1.000 ifl = 9.000  
instr 1: inumber = 9.000 idiv = 2.000 ifl = 4.000  
instr 1: inumber = 9.000 idiv = 3.000 ifl = 3.000  
instr 1: inumber = 9.000 idiv = 4.000 ifl = 2.000  
instr 1: inumber = 9.000 idiv = 5.000 ifl = 1.000  
instr 1: inumber = 9.000 idiv = 6.000 ifl = 1.000  
instr 1: inumber = 9.000 idiv = 7.000 ifl = 1.000  
instr 1: inumber = 9.000 idiv = 8.000 ifl = 1.000  
instr 1: inumber = 9.000 idiv = 9.000 ifl = 1.000  
instr 1: inumber = 9.000 idiv = 10.000 ifl = 0.000
```

## See Also

*abs, exp, int, log, log10, i, sqrt*

## Credits

Author: Istvan Varga  
New in Csound 5  
2005

# FLpack

FLpack — Provides the functionality of compressing and aligning FLTK widgets.

## Description

*FLpack* provides the functionality of compressing and aligning widgets.

## Syntax

```
FLpack iwidth, iheight, ix, iy, itype, ispace, iborder
```

## Initialization

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*itype* -- an integer number that modifies the appearance of the target widget.

The *itype* argument expresses the type of packing:

- 0 - vertical
- 1 - horizontal

*ispace* -- sets the space between the widgets.

*iborder* -- border type of the container. It is expressed by means of an integer number chosen from the following:

- 0 - no border
- 1 - down box border
- 2 - up box border
- 3 - engraved border
- 4 - embossed border
- 5 - black line border
- 6 - thin down border
- 7 - thin up border

## Performance

*FLpack* provides the functionality of compressing and aligning widgets.

Containers are useful to format the graphic appearance of the widgets. The most important container is *FLpanel*, that actually creates a window. It can be filled with other containers and/or valuators or other kinds of widgets.

There are no k-rate arguments in containers.

## Examples

The following example:

```

FLpanel "Panel1",450,300,100,100
FLpack 400,300, 10,40,0,15,3
gk1, ihs1      FLslider      "FLslider 1", 500, 1000, 2 ,1, -1, 300,15, 20,50
gk2, ihs2      FLslider      "FLslider 2", 300, 5000, 2 ,3, -1, 300,15, 20,100
gk3, ihs3      FLslider      "FLslider 3", 350, 1000, 2 ,5, -1, 300,15, 20,150
gk4, ihs4      FLslider      "FLslider 4", 250, 5000, 1 ,11, -1, 300,30, 20,200
gk5, ihs5      FLslider      "FLslider 5", 220, 8000, 2 ,1, -1, 300,15, 20,250
gk6, ihs6      FLslider      "FLslider 6", 1, 5000, 1 ,13, -1, 300,15, 20,300
gk7, ihs7      FLslider      "FLslider 7", 870, 5000, 1 ,15, -1, 300,30, 20,350
FLpackEnd
FLpanelEnd

```

...will produce this result, when resizing the window:



*FLpack*.

## See Also

*FLgroup, FLgroupEnd, FLpackEnd, FLpanel, FLpanelEnd, FLscroll, FLscrollEnd, FLtabs, FLtabsEnd*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLpackEnd

FLpackEnd — Marks the end of a group of compressed or aligned FLTK widgets.

## Description

Marks the end of a group of compressed or aligned FLTK widgets.

## Syntax

`FLpackEnd`

## Performance

Containers are useful to format the graphic appearance of the widgets. The most important container is *FLpanel*, that actually creates a window. It can be filled with other containers and/or valuators or other kinds of widgets.

There are no k-rate arguments in containers.

## See Also

*FLgroup*, *FLgroupEnd*, *FLpack*, *FLpanel*, *FLpanelEnd*, *FLscroll*, *FLscrollEnd*, *FLtabs*, *FLtabsEnd*

## Credits

Author: Gabriel Maldonado

New in version 4.22

## FLpack\_end

FLpack\_End — Marks the end of a group of compressed or aligned FLTK widgets.

### Description

Marks the end of a group of compressed or aligned FLTK widgets. This is another name for **FLpackEnd** provided for compatibility. See *FLpackEnd*

### Credits

Author: Gabriel Maldonado

New in version 4.22

# FLpanel

FLpanel — Creates a window that contains FLTK widgets.

## Description

Creates a window that contains FLTK widgets.

## Syntax

```
FLpanel "label", iwidth, iheight [, ix] [, iy] [, iborder] [, ikbdcapture] [, iclose]
```

## Initialization

*“label”* -- a double-quoted string containing some user-provided text, placed near the corresponding widget.

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* (optional) -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* (optional) -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iborder* (optional) -- border type of the container. It is expressed by means of an integer number chosen from the following:

- 0 - no border
- 1 - down box border
- 2 - up box border
- 3 - engraved border
- 4 - embossed border
- 5 - black line border
- 6 - thin down border
- 7 - thin up border

*ikbdcapture* (default = 0) -- If this flag is set to 1, keyboard events are captured by the window (for use with *sensekey* and *FLkeyIn*)

*iclose* (default = 0) -- If this flag is set to anything other than 0, the close button of the window is disabled, and the window cannot be closed by the user directly. It will close when *csound* exits.

## Performance

Containers are useful to format the graphic appearance of the widgets. The most important container is *FLpanel*, that actually creates a window. It can be filled with other containers and/or valuators or other kinds of widgets.

There are no k-rate arguments in containers.

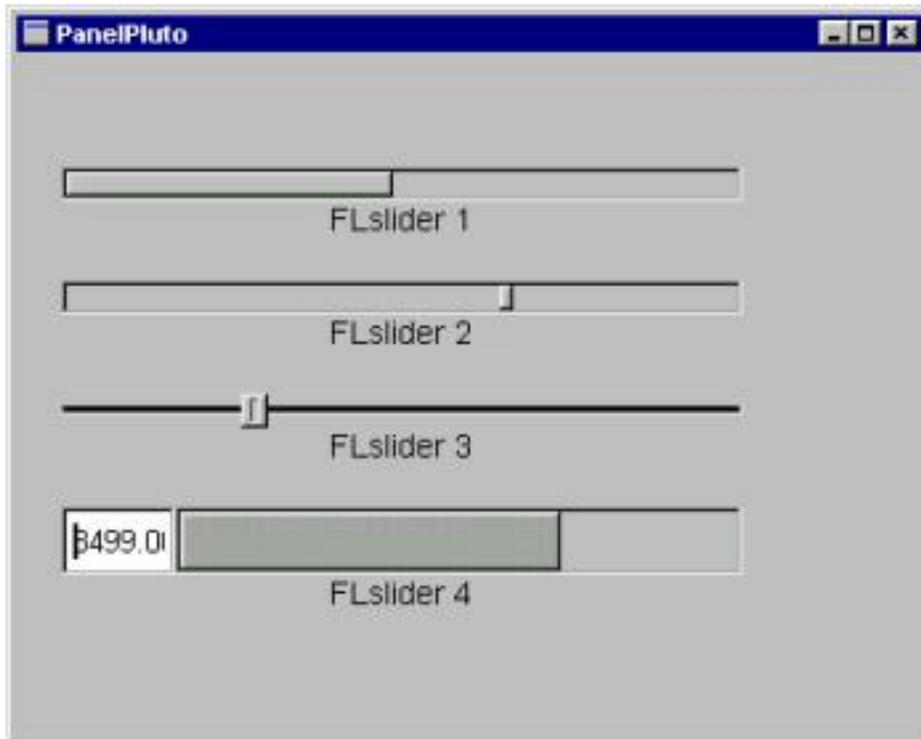
*FLpanel* creates a window. It must be followed by the opcode *FLpanelEnd* when all widgets internal to it are declared. For example:

```

        FLpanel    "PanelPluto",450,550,100,100 ;***** start of container
gk1, ih1 FLslider "FLslider 1", 500, 1000, 2 ,1, -1, 300,15, 20,50
gk2, ih2 FLslider "FLslider 2", 300, 5000, 2 ,3, -1, 300,15, 20,100
gk3, ih3 FLslider "FLslider 3", 350, 1000, 2 ,5, -1, 300,15, 20,150
gk4, ih4 FLslider "FLslider 4", 250, 5000, 1 ,11,-1, 300,30, 20,200
        FLpanelEnd ;***** end of container

```

will output the following result:



*FLpanel*.

If the *ikbdcapture* flag is set, the window captures keyboard events, and sends them to all *sensekey*. This flag modifies the behavior of *sensekey*, and makes it receive events from the FLTK window instead of *stdin*.

## Examples

Here is an example of the *FLpanel* opcode. It uses the file *FLpanel.csd* [examples/*FLpanel.csd*].

## Example 298. Example of the FLpanel opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o FLpanel.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Creates an empty window panel
sr = 44100
kr = 441
ksmps = 100
nchnls = 1

; Panel height in pixels
ipanelheight = 900
; Panel width in pixels
ipanelwidth = 400
; Horizontal position of the panel on screen in pixels
ix = 50
; Vertical position of the panel on screen in pixels
iy = 50

FLpanel "A Window Panel", ipanelheight, ipanelwidth, ix, iy
; End of panel contents
FLpanelEnd

;Run the widget thread!
FLrun

instr 1
endin

</CsInstruments>
<CsScore>

; 'Dummy' score event of 1 hour.
f 0 3600
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*FLgroup, FLgroupEnd, FLpack, FLpackEnd, FLpanelEnd, FLscroll, FLscrollEnd, FLtabs, FLtabsEnd, sensekey*

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.

# FLpanelEnd

FLpanelEnd — Marks the end of a group of FLTK widgets contained inside of a window (panel).

## Description

Marks the end of a group of FLTK widgets contained inside of a window (panel).

## Syntax

`FLpanelEnd`

## Performance

Containers are useful to format the graphic appearance of the widgets. The most important container is *FLpanel*, that actually creates a window. It can be filled with other containers and/or valuators or other kinds of widgets.

There are no k-rate arguments in containers.

## See Also

*FLgroup*, *FLgroupEnd*, *FLpack*, *FLpackEnd*, *FLpanel*, *FLscroll*, *FLscrollEnd*, *FLtabs*, *FLtabsEnd*

## Credits

Author: Gabriel Maldonado

New in version 4.22

## FLpanel\_end

FLpanel\_end — Marks the end of a group of FLTK widgets contained inside of a window (panel).

### Description

Marks the end of a group of FLTK widgets contained inside of a window (panel). This is another name for **FLpanelEnd** provided for compatibility. See *FLpanelEnd*

### Credits

Author: Gabriel Maldonado

New in version 4.22

# FLprintk

FLprintk — A FLTK opcode that prints a k-rate value at specified intervals.

## Description

*FLprintk* is similar to *printk* but shows values of a k-rate signal in a text field instead of on the console.

## Syntax

```
FLprintk itime, kval, idisp
```

## Initialization

*itime* -- how much time in seconds is to elapse between updated displays.

*idisp* -- a handle value that was output from a previous instance of the *FLvalue* opcode to display the current value of the current valuator in the *FLvalue* widget itself. If the user doesn't want to use this feature that displays current values, it must be set to a negative number by the user.

## Performance

*kval* -- k-rate signal to be displayed.

*FLprintk* is similar to *printk*, but shows values of a k-rate signal in a text field instead of showing it in the console. The *idisp* argument must be filled with the *ihandle* return value of a previous *FLvalue* opcode. While *FLvalue* should be placed in the header section of an orchestra inside an *FLpanel/FLpanelEnd* block, *FLprintk* must be placed inside an instrument to operate correctly. For this reason, it slows down performance and should be used for debugging purposes only.

## See Also

*FLbox*, *FLbutBank*, *FLbutton*, *FLprintk2*, *FLvalue*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLprintk2

FLprintk2 — A FLTK opcode that prints a new value every time a control-rate variable changes.

## Description

*FLprintk2* is similar to *FLprintk* but shows a k-rate variable's value only when it changes.

## Syntax

```
FLprintk2 kval, idisp
```

## Initialization

*idisp* -- a handle value that was output from a previous instance of the *FLvalue* opcode to display the current value of the current valuator in the *FLvalue* widget itself. If the user doesn't want to use this feature that displays current values, it must be set to a negative number by the user.

## Performance

*kval* -- k-rate signal to be displayed.

*FLprintk2* is similar to *FLprintk*, but shows the k-rate variable's value only each time it changes. Useful for monitoring MIDI control changes when using sliders. It should be used for debugging purposes only, since it slows-down performance.

## See Also

*FLbox*, *FLbutBank*, *FLbutton*, *FLprintk*, *FLvalue*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLroller

FLroller — A FLTK widget that creates a transversal knob.

## Description

*FLroller* is a sort of knob, but put transversally.

## Syntax

```
kout, ihandle FLroller "label", imin, imax, istep, iexp, itype, idisp, \  
    iwidth, iheight, ix, iy
```

## Initialization

*ihandle* -- a handle value (an integer number) that unequivocally references a corresponding widget. This is used by other opcodes that modify a widget's properties (see *Modifying FLTK Widget Appearance*). It is automatically output by *FLroller* and must not be set by the user label. (The user label is a double-quoted string containing some user-provided text placed near the widget.)

*"label"* -- a double-quoted string containing some user-provided text, placed near the corresponding widget.

*imin* -- minimum value of output range.

*imax* -- maximum value of output range.

*istep* -- a floating-point number indicating the increment of valuator value corresponding to of each mouse click. The *istep* argument allows the user to arbitrarily slow roller's motion, enabling arbitrary precision.

*iexp* -- an integer number denoting the behaviour of valuator:

- 0 = valuator output is linear
- -1 = valuator output is exponential

All other positive numbers for *iexp* indicate the number of an existing table that is used for indexing. Linear interpolation is provided in table indexing. A negative table number suppresses interpolation.



### IMPORTANT!

Notice that the tables used by valuator must be created with the *ftgen* opcode and placed in the orchestra before the corresponding valuator. They can not placed in the score. In fact, tables placed in the score are created later than the initialization of the opcodes placed in the header section of the orchestra.

*itype* -- an integer number denoting the appearance of the valuator.

The *itype* argument can be set to the following values:

- 1 - horizontal roller
- 2 - vertical roller

*idisp* -- a handle value that was output from a previous instance of the *FLvalue* opcode to display the current value of the current valuator in the *FLvalue* widget itself. If the user doesn't want to use this feature that displays current values, it must be set to a negative number by the user.

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

## Performance

*kout* -- output value

*FLroller* is a sort of knob, but put transversally:



FLroller.

## Examples

Here is an example of the FLroller opcode. It uses the file *FLroller.csd* [examples/FLroller.csd].

### Example 299. Example of the FLroller opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o FLroller.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; A sine with oscillator with flroller controlled frequency
sr = 44100
kr = 441
ksmps = 100
nchnls = 1

FLpanel "Frequency Roller", 900, 400, 50, 50
; Minimum value output by the roller
imin = 200
; Maximum value output by the roller
imax = 5000
; Increment with each pixel
istep = 1
; Logarithmic type roller selected
iexp = -1
```

```
    ; Roller graphic type (1=horizontal)
    itype = 1
    ; Display handle (-1=not used)
    idisp = -1
    ; Width of the roller in pixels
    iwidth = 300
    ; Height of the roller in pixels
    iheight = 50
    ; Distance of the left edge of the knob
    ; from the left edge of the panel
    ix = 300
    ; Distance of the top edge of the knob
    ; from the top edge of the panel
    iy = 50

    gkfreq, ihandle FLroller "Frequency", imin, imax, istep, iexp, itype, idisp, iwidth, iheight, ix, iy
; End of panel contents
FLpanelEnd
; Run the widget thread!
FLrun

instr 1
    iamp = 15000
    ifn = 1
    asig oscili iamp, gkfreq, ifn
    out asig
endin

</CsInstruments>
<CsScore>

; Function table that defines a single cycle
; of a sine wave.
f 1 0 1024 10 1

; Instrument 1 will play a note for 1 hour.
i 1 0 3600
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*FLcount, FLjoy, FLknob, FLslider, FLtext*

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.

# FLrun

FLrun — Starts the FLTK widget thread.

## Description

Starts the FLTK widget thread.

## Syntax

`FLrun`

## Performance

This opcode must be located at the end of all widget declarations. It has no arguments, and its purpose is to start the thread related to widgets. Widgets would not operate if *FLrun* is missing.

## See Also

*FLgetsnap, FLloadsnap, FLsavesnap, FLsetsnap, FLupdate*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsavesnap

FLsavesnap — Saves all snapshots currently created into a file.

## Description

*FLsavesnap* saves all snapshots currently created (i.e. the entire memory bank) into a file.

## Syntax

```
FLsavesnap "filename" [, igroup]
```

## Initialization

*“filename”* -- a double-quoted string corresponding to a file to store a bank of snapshots.

*igroup* -- (optional) an integer number referring to a snapshot-related group of widget. It allows to get/set, or to load/save the state of a subset of valuator. Default value is zero that refers to the first group. The group number is determined by the opcode *FLsetSnapGroup*.



### Note

The *igroup* parameter has not been yet fully implemented in the current version of csound. Please do not rely on it yet.

## Performance

*FLsavesnap* saves all snapshots currently created (i.e. the entire memory bank) into a file whose name is *filename*. Since the file is a text file, snapshot values can also be edited manually by means of a text editor. The format of the data stored in the file is the following (at present time, this could be changed in next Csound version):

```
----- 0 -----
FLvalue 0 0 1 0 ""
FLvalue 0 0 1 0 ""
FLvalue 0 0 1 0 ""
FLslider 331.946 80 5000 -1 "frequency of the first oscillator"
FLslider 385.923 80 5000 -1 "frequency of the second oscillator"
FLslider 80 80 5000 -1 "frequency of the third oscillator"
FLcount 0 0 10 0 "this index must point to the location number where snapshot is stored"
FLbutton 0 0 1 0 "Store snapshot to current index"
FLbutton 0 0 1 0 "Save snapshot bank to disk"
FLbutton 0 0 1 0 "Load snapshot bank from disk"
FLbox 0 0 1 0 ""
----- 1 -----
FLvalue 0 0 1 0 ""
FLvalue 0 0 1 0 ""
FLvalue 0 0 1 0 ""
FLslider 819.72 80 5000 -1 "frequency of the first oscillator"
FLslider 385.923 80 5000 -1 "frequency of the second oscillator"
FLslider 80 80 5000 -1 "frequency of the third oscillator"
FLcount 1 0 10 0 "this index must point to the location number where snapshot is stored"
FLbutton 0 0 1 0 "Store snapshot to current index"
FLbutton 0 0 1 0 "Save snapshot bank to disk"
FLbutton 0 0 1 0 "Load snapshot bank from disk"
FLbox 0 0 1 0 ""
```

```

----- 2 -----
..... etc...
----- 3 -----
..... etc...
-----

```

As you can see, each snapshot contain several lines. Each snapshot is separated from previous and next snapshot by a line of this kind:

```
"----- snapshot Num -----"
```

Then there are several lines containing data. Each of these lines corresponds to a widget.

The first field of each line is an unquoted string containing opcode name corresponding to that widget. Second field is a number that expresses current value of a snapshot. In current version, this is the only field that can be modified manually. The third and fourth fields shows minimum and maximum values allowed for that valuator. The fifth field is a special number that indicates if the valuator is linear (value 0), exponential (value -1), or is indexed by a table interpolating values (negative table numbers) or non-interpolating (positive table numbers). The last field is a quoted string with the label of the widget. Last line of the file is always

```
"-----"
```

.

Note that *FLvalue* and *FLbox* are not valuators and their values are fixed, so they cannot be modified.

For purposes of snapshot saving, widgets can be grouped, so that snapshots affect only a defined group of widgets. The opcode *FLsetSnapGroup* is used to specify the group for all widgets declared after it, until the next *FLsetSnapGroup* statement.

## Examples

Here is a simple example of the FLTK snapshot saving. It uses the file *FLsavesnap\_simple.csd* [examples/FLsavesnap\_simple.csd].

### Example 300. Example of FLTK snapshot saving.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc    ;;RT audio I/O
</CsOptions>
<CsInstruments>

sr=48000
ksmps=128
nchnls=2

; Example by Hector Centeno and Andres Cabrera 2007

```

```

; giSWMtab4 ftgen 0, 0, 513, 21, 10, 1, .3
; giSWMtab4M ftgen 0, 0, 64, 7, 1, 50, 1

FLpanel "Snapshots", 530, 190, 40, 410, 3
  FLcolor 100, 118, 140
  ivalSM1 FLvalue "", 70, 20, 270, 20
  gksliderA, gislidSM1 FLslider "Slider", -4, 4, 0, 3, ivalSM1, 250, 20, 20, 20
  itext1 FLbox "store", 1, 1, 14, 50, 25, 355, 15
  itext2 FLbox "load", 1, 1, 14, 50, 25, 415, 15
  gksnap, ibuttn1 FLbutton "1", 1, 0, 11, 25, 25, 364, 45, 0, 3, 0, 3, 1
  gksnap, ibuttn2 FLbutton "2", 1, 0, 11, 25, 25, 364, 75, 0, 3, 0, 3, 2
  gksnap, ibuttn3 FLbutton "3", 1, 0, 11, 25, 25, 364, 105, 0, 3, 0, 3, 3
  gksnap, ibuttn4 FLbutton "4", 1, 0, 11, 25, 25, 364, 135, 0, 3, 0, 3, 4

  gkload, ibuttn1 FLbutton "1", 1, 0, 11, 25, 25, 424, 45, 0, 4, 0, 3, 1
  gkload, ibuttn2 FLbutton "2", 1, 0, 11, 25, 25, 424, 75, 0, 4, 0, 3, 2
  gkload, ibuttn3 FLbutton "3", 1, 0, 11, 25, 25, 424, 105, 0, 4, 0, 3, 3
  gkload, ibuttn4 FLbutton "4", 1, 0, 11, 25, 25, 424, 135, 0, 4, 0, 3, 4

  ivalSM2 FLvalue "", 70, 20, 270, 80
  gkknobA, gislidSM2 FLknob "Knob", -4, 4, 0, 3, ivalSM2, 60, 120, 60
FLpanelEnd
FLsetVal_i 1, gislidSM1
FLsetVal_i 1, gislidSM2
FLrun

instr 1

endin

instr 3 ; Save snapshot
index init 0
ipstno = p4
Sfile sprintf "snapshot_simple.%d.snap", ipstno

inumsnap, inumval FLsetsnap index ;, -1, igroup
FLsavesnap Sfile

endin

instr 4 ;Load snapshot
index init 0
ipstno = p4
Sfile sprintf "snapshot_simple.%d.snap", ipstno

FLloadsnap Sfile
inumload FLgetsnap index ;, igroup

endin

</CsInstruments>

<CsScore>
f 0 3600

e

</CsScore>

</CsoundSynthesizer>

```

Here is another example of FLTK snapshot saving using snapshot groups. It uses the file *FLsavesnap.csd* [examples/FLsavesnap.csd].

**Example 301. Example of FLTK snapshot saving using snapshot groups.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsSoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in
-odac         -iadc      ;;RT audio I/O
</CsOptions>
<CsInstruments>

sr=48000
ksmps=128
nchnls=2

; Example by Hector Centeno and Andres Cabrera 2007

; giSWMtab4 ftgen 0, 0, 513, 21, 10, 1, .3
; giSWMtab4M ftgen 0, 0, 64, 7, 1, 50, 1

FLpanel "Snapshots", 530, 350, 40, 410, 3
  FLcolor 100, 118 ,140
  FLsetSnapGroup 0
  ivalSM1   FLvalue  "", 70, 20, 270, 20
  ivalSM2   FLvalue  "", 70, 20, 270, 60
  ivalSM3   FLvalue  "", 70, 20, 270, 100
  ivalSM4   FLvalue  "", 70, 20, 270, 140
  gksliderA, gislidSM1  FLslider "Slider A", -4, 4, 0, 3, ivalSM1, 250, 20, 20, 20
  gksliderB, gislidSM2  FLslider "Slider B", 1, 10, 0, 3, ivalSM2, 250, 20, 20, 60
  gksliderC, gislidSM3  FLslider "Slider C", 0, 1, 0, 3, ivalSM3, 250, 20, 20, 100
  gksliderD, gislidSM4  FLslider "Slider D", 0, 1, 0, 3, ivalSM4, 250, 20, 20, 140
  itext1    FLbox    "store", 1, 1, 14, 50, 25, 355, 15
  itext2    FLbox    "load", 1, 1, 14, 50, 25, 415, 15
  itext3    FLbox    "Group 1", 1, 1, 14, 30, 145, 485, 15
  gksnap, ibuttn1  FLbutton "1", 1, 0, 11, 25, 25, 364, 45, 0, 3, 0, 3, 1
  gksnap, ibuttn2  FLbutton "2", 1, 0, 11, 25, 25, 364, 75, 0, 3, 0, 3, 2
  gksnap, ibuttn3  FLbutton "3", 1, 0, 11, 25, 25, 364, 105, 0, 3, 0, 3, 3
  gksnap, ibuttn4  FLbutton "4", 1, 0, 11, 25, 25, 364, 135, 0, 3, 0, 3, 4
  gkload, ibuttn1  FLbutton "1", 1, 0, 11, 25, 25, 424, 45, 0, 4, 0, 3, 1
  gkload, ibuttn2  FLbutton "2", 1, 0, 11, 25, 25, 424, 75, 0, 4, 0, 3, 2
  gkload, ibuttn3  FLbutton "3", 1, 0, 11, 25, 25, 424, 105, 0, 4, 0, 3, 3
  gkload, ibuttn4  FLbutton "4", 1, 0, 11, 25, 25, 424, 135, 0, 4, 0, 3, 4

  FLcolor 100, 140 ,118
  FLsetSnapGroup 1
  ivalSM5   FLvalue  "", 70, 20, 270, 190
  ivalSM6   FLvalue  "", 70, 20, 270, 230
  ivalSM7   FLvalue  "", 70, 20, 270, 270
  ivalSM8   FLvalue  "", 70, 20, 270, 310
  gkknobA, gislidSM5  FLknob "Knob A", -4, 4, 0, 3, ivalSM5, 45, 10, 230
  gkknobB, gislidSM6  FLknob "Knob B", 1, 10, 0, 3, ivalSM6, 45, 75, 230
  gkknobC, gislidSM7  FLknob "Knob C", 0, 1, 0, 3, ivalSM7, 45, 140, 230
  gkknobD, gislidSM8  FLknob "Knob D", 0, 1, 0, 3, ivalSM8, 45, 205, 230
  itext4    FLbox    "store", 1, 1, 14, 50, 25, 355, 185
  itext5    FLbox    "load", 1, 1, 14, 50, 25, 415, 185
  itext6    FLbox    "Group 2", 1, 1, 14, 30, 145, 485, 185
  gksnap, ibuttn1  FLbutton "5", 1, 0, 11, 25, 25, 364, 215, 0, 3, 0, 3, 5
  gksnap, ibuttn2  FLbutton "6", 1, 0, 11, 25, 25, 364, 245, 0, 3, 0, 3, 6
  gksnap, ibuttn3  FLbutton "7", 1, 0, 11, 25, 25, 364, 275, 0, 3, 0, 3, 7
  gksnap, ibuttn4  FLbutton "8", 1, 0, 11, 25, 25, 364, 305, 0, 3, 0, 3, 8
  gkload, ibuttn1  FLbutton "5", 1, 0, 11, 25, 25, 424, 215, 0, 4, 0, 3, 5
  gkload, ibuttn2  FLbutton "6", 1, 0, 11, 25, 25, 424, 245, 0, 4, 0, 3, 6
  gkload, ibuttn3  FLbutton "7", 1, 0, 11, 25, 25, 424, 275, 0, 4, 0, 3, 7
  gkload, ibuttn4  FLbutton "8", 1, 0, 11, 25, 25, 424, 305, 0, 4, 0, 3, 8

```

```

FLpanelEnd
FLsetVal_i 1, gislidSM1
FLsetVal_i 1, gislidSM2
FLsetVal_i 0, gislidSM3
FLsetVal_i 0, gislidSM4
FLsetVal_i 1, gislidSM5
FLsetVal_i 1, gislidSM6
FLsetVal_i 0, gislidSM7
FLsetVal_i 0, gislidSM8
FLrun

instr 1

endin

instr 3 ; Save snapshot
index init 0
ipstno = p4
igroup = 0
Sfile sprintf "PVCsynth.%d.snap", ipstno
if ipstno > 4 then
    igroup = 1
endif

inumsnap, inumval FLsetsnap index , -1, igroup
FLsavesnap Sfile

endin

instr 4 ;Load snapshot
index init 0
ipstno = p4
igroup = 0
Sfile sprintf "PVCsynth.%d.snap", ipstno
if ipstno > 4 then
    igroup = 1
endif

FLloadsnap Sfile
inumload FLgetsnap index , igroup

endin

</CsInstruments>

<CsScore>
    ;Dummy table for FLgetsnap
    ; f 1 0 1024 10 1
    f 0 3600

e

</CsScore>

</CsoundSynthesizer>

```

## See Also

*FLgetsnap, FLloadsnap, FLsetSnapGroup, FLrun, FLsetsnap, FLupdate*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLscroll

FLscroll — A FLTK opcode that adds scroll bars to an area.

## Description

*FLscroll* adds scroll bars to an area.

## Syntax

```
FLscroll iwidth, iheight [, ix] [, iy]
```

## Initialization

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* (optional) -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* (optional) -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

## Performance

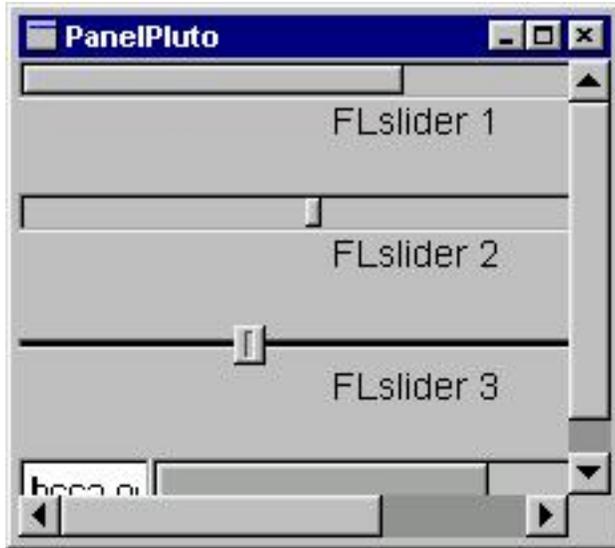
Containers are useful to format the graphic appearance of the widgets. The most important container is *FLpanel*, that actually creates a window. It can be filled with other containers and/or valuators or other kinds of widgets.

There are no k-rate arguments in containers.

*FLscroll* adds scroll bars to an area. Normally you must set arguments *iwidth* and *iheight* equal to that of the parent window or other parent container. *ix* and *iy* are optional since they normally are set to zero. For example the following code:

```
        FLpanel      "PanelPluto", 400, 300, 100, 100
        FLscroll    400, 300
gk1, ih1 FLslider  "FLslider 1", 500, 1000, 2, 1, -1, 300, 15, 20, 50
gk2, ih2 FLslider  "FLslider 2", 300, 5000, 2, 3, -1, 300, 15, 20, 100
gk3, ih3 FLslider  "FLslider 3", 350, 1000, 2, 5, -1, 300, 15, 20, 150
gk4, ih4 FLslider  "FLslider 4", 250, 5000, 1, 11, -1, 300, 30, 20, 200
        FLscrollEnd
        FLpanelEnd
```

will show scroll bars, when the main window size is reduced:



FLscroll.

## Examples

Here is an example of the FLscroll opcode. It uses the file *FLscroll.csd* [examples/FLscroll.csd].

### Example 302. Example of the FLscroll opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc    -d      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o FLscroll.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Demonstration of the flscroll opcode which enables
; the use of widget sizes and placings beyond the
; dimensions of the containing panel
sr = 44100
kr = 441
ksmps = 100
nchnls = 1

FLpanel "Text Box", 420, 200, 50, 50
  iwidth = 420
  iheight = 200
  ix = 0
  iy = 0
  FLscroll iwidth, iheight, ix, iy
  ih3 FLbox "DRAG THE SCROLL BAR TO THE RIGHT IN ORDER TO READ THE REST OF THIS TEXT!", 1, 10, 20, 87
  FLscrollEnd
; End of panel contents
FLpanelEnd
; Run the widget thread!
```

```
FLrun

instr 1
endin

</CsInstruments>
<CsScore>

; 'Dummy' score event of 1 hour.
f 0 3600
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*FLgroup, FLgroupEnd, FLpack, FLpackEnd, FLpanel, FLpanelEnd, FLscrollEnd, FLtabs, FLtabsEnd*

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.

# FLscrollEnd

FLscrollEnd — A FLTK opcode that marks the end of an area with scrollbars.

## Description

A FLTK opcode that marks the end of an area with scrollbars.

## Syntax

```
FLscrollEnd
```

## Performance

Containers are useful to format the graphic appearance of the widgets. The most important container is *FLpanel*, that actually creates a window. It can be filled with other containers and/or valuators or other kinds of widgets.

There are no k-rate arguments in containers.

## See Also

*FLgroup*, *FLgroupEnd*, *FLpack*, *FLpackEnd*, *FLpanel*, *FLpanelEnd*, *FLscroll*, *FLtabs*, *FLtabsEnd*

## Credits

Author: Gabriel Maldonado

New in version 4.22

## FLscroll\_end

FLscroll\_end — A FLTK opcode that marks the end of an area with scrollbars.

### Description

A FLTK opcode that marks the end of an area with scrollbars. This is another name for **FLscrollEnd** provided for compatibility. See *FLscrollEnd*

### Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetAlign

FLsetAlign — Sets the text alignment of a label of a FLTK widget.

## Description

*FLsetAlign* sets the text alignment of the label of the target widget.

## Syntax

```
FLsetAlign ialign, ihandle
```

## Initialization

*ialign* -- sets the alignment of the label text of widgets.

The legal values for the *ialign* argument are:

- 1 - align center
- 2 - align top
- 3 - align bottom
- 4 - align left
- 5 - align right
- 6 - align top-left
- 7 - align top-right
- 8 - align bottom-left
- 9 - align bottom-right

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetBox

FLsetBox — Sets the appearance of a box surrounding a FLTK widget.

## Description

*FLsetBox* sets the appearance of a box surrounding the target widget.

## Syntax

```
FLsetBox itype, ihandle
```

## Initialization

*itype* -- an integer number that modify the appearance of the target widget.

Legal values for the *itype* argument are:

- 1 - flat box
- 2 - up box
- 3 - down box
- 4 - thin up box
- 5 - thin down box
- 6 - engraved box
- 7 - embossed box
- 8 - border box
- 9 - shadow box
- 10 - rounded box
- 11 - rounded box with shadow
- 12 - rounded flat box
- 13 - rounded up box
- 14 - rounded down box
- 15 - diamond up box
- 16 - diamond down box
- 17 - oval box
- 18 - oval shadow box
- 19 - oval flat box

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetColor

FLsetColor — Sets the primary color of a FLTK widget.

## Description

*FLsetColor* sets the primary color of the target widget.

## Syntax

```
FLsetColor ired, igreen, iblue, ihandle
```

## Initialization

*ired* -- The red color of the target widget. The range for each RGB component is 0-255

*igreen* -- The green color of the target widget. The range for each RGB component is 0-255

*iblue* -- The blue color of the target widget. The range for each RGB component is 0-255

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## Examples

Here is an example of the FLsetcolor opcode. It uses the file *FLsetcolor.csd* [examples/FLsetcolor.csd].

### Example 303. Example of the FLsetcolor opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o FLsetcolor.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Using the opcode flsetcolor to change from the
; default colours for widgets
sr = 44100
kr = 441
ksmps = 100
nchnls = 1

FLpanel "Coloured Sliders", 900, 360, 50, 50
gkfreq, ihandle FLslider "A Red Slider", 200, 5000, -1, 5, -1, 750, 30, 85, 50
ired1 = 255
igreen1 = 0
```

```
ibblue1 = 0
FLsetColor ired1, igreen1, ibblue1, ihandle

gkfreq, ihandle FLslider "A Green Slider", 200, 5000, -1, 5, -1, 750, 30, 85, 150
ired1 = 0
igreen1 = 255
ibblue1 = 0
FLsetColor ired1, igreen1, ibblue1, ihandle

gkfreq, ihandle FLslider "A Blue Slider", 200, 5000, -1, 5, -1, 750, 30, 85, 250
ired1 = 0
igreen1 = 0
ibblue1 = 255
FLsetColor ired1, igreen1, ibblue1, ihandle
; End of panel contents
FLpanelEnd
; Run the widget thread!
FLrun

instr 1
endin

</CsInstruments>
<CsScore>

; 'Dummy' score event for 1 hour.
f 0 3600
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*FLcolor2, FLhide, FLlabel, FLsetAlign, FLsetBox, FLsetColor, FLsetColor2, FLsetFont, FLsetPosition, FLsetSize, FLsetText, FLsetTextColor, FLsetTextSize, FLsetTextType, FLsetVal\_i, FLsetVal, FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.

# FLsetColor2

FLsetColor2 — Sets the secondary (or selection) color of a FLTK widget.

## Description

*FLsetColor2* sets the secondary (or selection) color of the target widget.

## Syntax

```
FLsetColor2 ired, igrreen, iblue, ihandle
```

## Initialization

*ired* -- The red color of the target widget. The range for each RGB component is 0-255

*igrreen* -- The green color of the target widget. The range for each RGB component is 0-255

*iblue* -- The blue color of the target widget. The range for each RGB component is 0-255

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetFont

FLsetFont — Sets the font type of a FLTK widget.

## Description

*FLsetFont* sets the font type of the target widget.

## Syntax

```
FLsetFont ifont, ihandle
```

## Initialization

*ifont* -- sets the the font type of the label of a widget.

Legal values for ifont argument are:

- 1 - Helvetica (same as Arial under Windows)
- 2 - Helvetica Bold
- 3 - Helvetica Italic
- 4 - Helvetica Bold Italic
- 5 - Courier
- 6 - Courier Bold
- 7 - Courier Italic
- 8 - Courier Bold Italic
- 9 - Times
- 10 - Times Bold
- 11 - Times Italic
- 12 - Times Bold Italic
- 13 - Symbol
- 14 - Screen
- 15 - Screen Bold
- 16 - Dingbats

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## See Also

*FLcolor2, FLhide, FLlabel, FLsetAlign, FLsetBox, FLsetColor, FLsetColor2, FLsetFont, FLsetPosition, FLsetSize, FLsetText, FLsetTextColor, FLsetTextSize, FLsetTextType, FLsetVal\_i, FLsetVal, FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetPosition

FLsetPosition — Sets the position of a FLTK widget.

## Description

*FLsetPosition* sets the position of the target widget according to the *ix* and *iy* arguments.

## Syntax

```
FLsetPosition ix, iy, ihandle
```

## Initialization

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetSize

FLsetSize — Resizes a FLTK widget.

## Description

*FLsetSize* resizes the target widget (not the size of its text) according to the *iwidth* and *iheight* arguments.

## Syntax

```
FLsetSize iwidth, iheight, ihandle
```

## Initialization

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetsnap

FLsetsnap — Stores the current status of all FLTK valuators into a snapshot location.

## Description

*FLsetsnap* stores the current status of all valuators present in the orchestra into a snapshot location (in memory).

## Syntax

```
inumsnap, inumval FLsetsnap index [, ifn, igroup]
```

## Initialization

*inumsnap* -- current number of snapshots.

*inumval* -- number of valuators (whose value is stored in a snapshot) present in current orchestra.

*index* -- a number referring unequivocally to a snapshot. Several snapshots can be stored in the same bank.

*ifn* (optional) -- optional argument referring to an already allocated table, to store values of a snapshot.

*igroup* -- (optional) an integer number referring to a snapshot-related group of widget. It allows to get/set, or to load/save the state of a subset of valuators. Default value is zero that refers to the first group. The group number is determined by the opcode *FLsetSnapGroup*.



### Note

The *igroup* parameter has not been yet fully implemented in the current version of csound. Please do not rely on it yet.

## Performance

The *FLsetsnap* opcode stores current status of all valuators present in the orchestra into a snapshot location (in memory). Any number of snapshots can be stored in the current bank. Banks are structures that only exist in memory, there are no other reference to them other that they can be accessed by *FLsetsnap*, *FLsavesnap*, *FLloadsnap* and *FLgetsnap* opcodes. Only a single bank can be present in memory.

If the optional *ifn* argument refers to an already allocated and valid table, the snapshot will be stored in the table instead of in the bank. So that table can be accessed from other Csound opcodes.

The *index* argument unequivocally refers to a determinate snapshot. If the value of *index* refers to a previously stored snapshot, all its old values will be replaced with current ones. If *index* refers to a snapshot that doesn't exist, a new snapshot will be created. If the *index* value is not adjacent with that of a previously created snapshot, some empty snapshots will be created. For example, if a location with *index* 0 contains the only and unique snapshot present in a bank and the user stores a new snapshot using *index* 5, all locations between 1 and 4 will automatically contain empty snapshots. Empty snapshots don't contain any data and are neutral.

*FLsetsnap* outputs the current number of snapshots (the *inumsnap* argument) and the total number of values stored in each snapshot (*inumval*). *inumval* is equal to the number of valuators present in the orchestra.

For purposes of snapshot saving, widgets can be grouped, so that snapshots affect only a defined group of widgets. The opcode *FLsetSnapGroup* is used to specify the group for all widgets declared after it, until the next *FLsetSnapGroup* statement.

## See Also

*FLgetsnap*, *FLloadsnap*, *FLsetSnapGroup*, *FLrun*, *FLsavesnap*, *FLupdate*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetSnapGroup

FLsetSnapGroup — Determines the snapshot group for FL valuator.

## Description

*FLsetSnapGroup* determines the snapshot group of valuator declared after it.

## Syntax

```
FLsetSnapGroup igroup
```

## Initialization

*igroup* -- (optional) an integer number referring to a snapshot-related group of widget. It allows to get/set, or to load/save the state of a subset of valuator.



### Note

The *igroup* parameter has not been yet fully implemented in the current version of csound. Please do not rely on it yet.

## Performance

For purposes of snapshot saving, widgets can be grouped, so that snapshots affect only a defined group of widgets. The opcode *FLsetSnapGroup* is used to specify the group for all widgets declared after it, until the next *FLsetSnapGroup* statement.

*FLsetSnapGroup* determines the snapshot group of a declared valuator. To make a valuator belong to a stated group, you have to place *FLsetSnapGroup* just before the declaration of the widget itself. The group stated by *FLsetSnapGroup* lasts for all valuator declared after it, until a new *FLsetSnapGroup* statement with a different group is encountered. If no *FLsetSnapGroup* statement are present in an orchestra, the default group for all widgets will be group zero.

## See Also

*FLgetsnap*, *FLsetsnap*, *FLloadsnap*, *FLsavesnap*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetText

FLsetText — Sets the label of a FLTK widget.

## Description

*FLsetText* sets the label of the target widget to the double-quoted text string provided with the *itext* argument.

## Syntax

```
FLsetText "itext", ihandle
```

## Initialization

*“itext”* -- a double-quoted string denoting the text of the label of the widget.

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## Examples

Here is an example of the FLsetText opcode. It uses the file *FLsetText.csd* [examples/FLsetText.csd].

### Example 304. Example of the FLsetText opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o FLsetText.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 128
nchnls = 2

; Example by Giorgio Zucco and Andres Cabrera 2007

FLpanel "FLsetText",250,100,50,50

gk1,giha FLcount " ", 1, 20, 1, 20, 1, 200, 40, 20, 20, 0, 1, 0, 1

FLpanelEnd
FLrun
```

```
    instr 1
; This instrument is triggered by FLcount above each time
; its value changes
iname = i(gk1)
print iname
; Must use FLsetText on the init pass!
if (iname == 1) igoto text1
if (iname == 2) igoto text2
if (iname == 3) igoto text3

igoto end

text1:
FLsetText "FM",giha
igoto end

text2:
FLsetText "GRANUL",giha
igoto end

text3:
FLsetText "PLUCK",giha
igoto end

end:
    endin

</CsInstruments>
<CsScore>

f 0 3600

</CsScore>
</CsoundSynthesizer>
```

## See Also

*FLcolor2, FLhide, FLlabel, FLsetAlign, FLsetBox, FLsetColor, FLsetColor2, FLsetFont, FLsetPosition, FLsetSize, FLsetText, FLsetTextColor, FLsetTextSize, FLsetTextType, FLsetVal\_i, FLsetVal, FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetTextColor

FLsetTextColor — Sets the color of the text label of a FLTK widget.

## Description

*FLsetTextColor* sets the color of the text label of the target widget.

## Syntax

```
FLsetTextColor ired, iblue, igreen, ihandle
```

## Initialization

*ired* -- The red color of the target widget. The range for each RGB component is 0-255

*igreen* -- The green color of the target widget. The range for each RGB component is 0-255

*iblue* -- The blue color of the target widget. The range for each RGB component is 0-255

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetTextSize

FLsetTextSize — Sets the size of the text label of a FLTK widget.

## Description

*FLsetTextSize* sets the size of the text label of the target widget.

## Syntax

```
FLsetTextSize isize, ihandle
```

## Initialization

*isize* -- size of the font of the target widget. Normal values are in the order of 15. Greater numbers enlarge font size, while smaller numbers reduce it.

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetTextType

FLsetTextType — Sets some font attributes of the text label of a FLTK widget.

## Description

*FLsetTextType* sets some attributes related to the fonts of the text label of the target widget.

## Syntax

```
FLsetTextType itype, ihandle
```

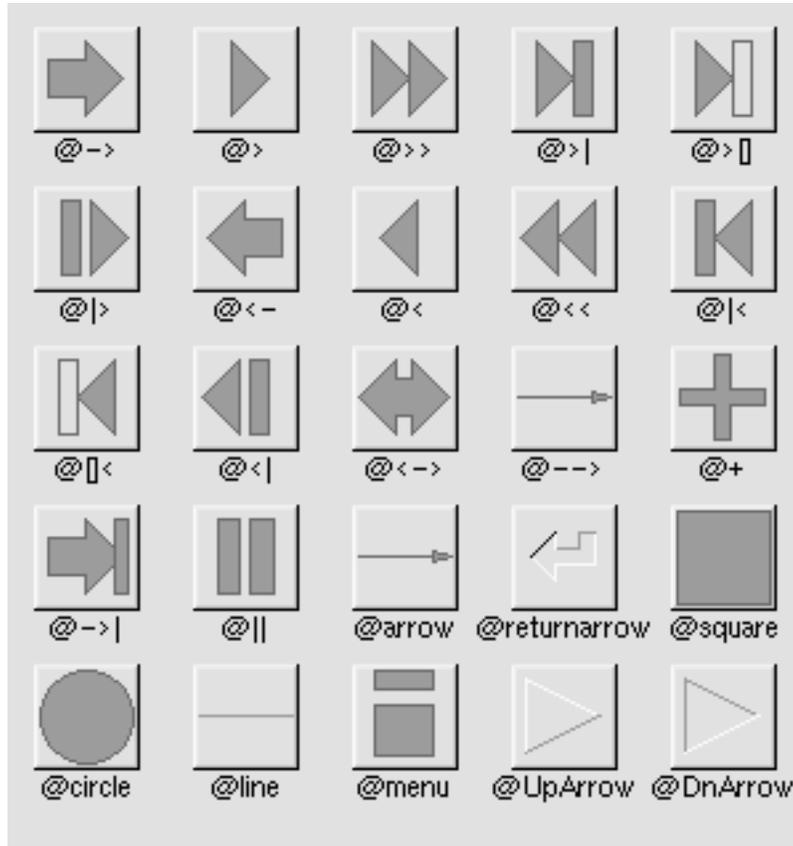
## Initialization

*itype* -- an integer number that modify the appearance of the target widget.

The legal values of *itype* are:

- 0 - normal label
- 1 - no label (hides the text)
- 2 - symbol label (see below)
- 3 - shadow label
- 4 - engraved label
- 5- embossed label
- 6- bitmap label (not implemented yet)
- 7- pixmap label (not implemented yet)
- 8- image label (not implemented yet)
- 9- multi label (not implemented yet)
- 10- free-type label (not implemented yet)

When using *itype*=3 (symbol label), it is possible to assign a graphical symbol instead of the text label of the target widget. In this case, the string of the target label must always start with “@”. If it starts with something else (or the symbol is not found), the label is drawn normally. The following symbols are supported:



FLTK label supported symbols.

The @ sign may be followed by the following optional “formatting” characters, in this order:

1. “#” forces square scaling rather than distortion to the widget's shape.
2. +[1-9] or -[1-9] tweaks the scaling a little bigger or smaller.
3. [1-9] rotates by a multiple of 45 degrees. “6” does nothing, the others point in the direction of that key on a numeric keypad.

Notice that with *FLbox* and *FLbutton*, it is not necessary to call *FLsetTextType* opcode at all in order to use a symbol. In this case, it is sufficient to set a label starting with “@” followed by the proper formatting string.

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetVal\_i

FLsetVal\_i — Sets the value of a FLTK valuator to a number provided by the user.

## Description

*FLsetVal\_i* forces the value of a valuator to a number provided by the user.

## Syntax

```
FLsetVal_i ivalue, ihandle
```

## Initialization

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## Performance

*ivalue* -- Value to set the widget to.



### Note

*FLsetVal* is not fully implemented yet, and may crash in certain cases (e.g. when setting the value of a widget connected to a *FLvalue* widget- in this case use two separate *FLsetVal\_i*).

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetVal

FLsetVal — Sets the value of a FLTK valuator at control-rate.

## Description

*FLsetVal* is almost identical to *FLsetVal\_i*. Except it operates at k-rate and it affects the target valuator only when *ktrig* is set to a non-zero value.

## Syntax

```
FLsetVal ktrig, kvalue, ihandle
```

## Initialization

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## Performance

*ktrig* -- triggers the opcode when different than 0.

*kvalue* -- Value to set the widget to.



### Note

*FLsetVal* is not fully implemented yet, and may crash in certain cases (e.g. when setting the value of a widget connected to a *FLvalue* widget- in this case use two separate *FLsetVal*)

## See Also

*FLcolor*, *FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLshow

FLshow — Restores the visibility of a previously hidden FLTK widget.

## Description

*FLshow* restores the visibility of a previously hidden widget.

## Syntax

```
FLshow ihandle
```

## Initialization

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLslidBnk

FLslidBnk — A FLTK widget containing a bank of horizontal sliders.

## Description

*FLslidBnk* is a widget containing a bank of horizontal sliders.

## Syntax

```
FLslidBnk "names", inumsliders [, ioutable] [, iwidth] [, iheight] [, ix] \  
[, iy] [, itypetable] [, iexptable] [, istart_index] [, iminmaxtable]
```

## Initialization

*"names"* -- a double-quoted string containing the names of each slider. Each slider can have a different name. Separate each name with "@" character, for example: "frequency@amplitude@cutoff". It is possible to not provide any name by giving a single space " ". In this case, the opcode will automatically assign a progressive number as a label for each slider.

*inumsliders* -- the number of sliders.

*ioutable* (optional, default=0) -- number of a previously-allocated table in which to store output values of each slider. The user must be sure that table size is large enough to contain all output cells, otherwise a segfault will crash Csound. By assigning zero to this argument, the output will be directed to the zak space in the k-rate zone. In this case, the zak space must be previously allocated with the *zakinit* opcode and the user must be sure that the allocation size is big enough to cover all sliders. The default value is zero (i.e. store output in zak space).

*istart\_index* (optional, default=0) -- an integer number referring to a starting offset of output cell locations. It can be positive to allow multiple banks of sliders to output in the same table or in the zak space. The default value is zero (no offset).

*iminmaxtable* (optional, default=0) -- number of a previously-defined table containing a list of min-max pairs, referred to each slider. A zero value defaults to the 0 to 1 range for all sliders without necessity to provide a table. The default value is zero.

*iexptable* (optional, default=0) -- number of a previously-defined table containing a list of identifiers (i.e. integer numbers) provided to modify the behaviour of each slider independently. Identifiers can assume the following values:

- -1 -- exponential curve response
- 0 -- linear response
- number > than 0 -- follow the curve of a previously-defined table to shape the response of the corresponding slider. In this case, the number corresponds to table number.

You can assume that all sliders of the bank have the same response curve (exponential or linear). In this case, you can assign -1 or 0 to *iexptable* without worrying about previously defining any table. The default value is zero (all sliders have a linear response, without having to provide a table).

*ityetable* (optional, default=0) -- number of a previously-defined table containing a list of identifiers (i.e. integer numbers) provided to modify the aspect of each individual slider independently. Identifiers can assume the following values:

- 0 = Nice slider
- 1 = Fill slider
- 3 = Normal slider
- 5 = Nice slider
- 7 = Nice slider with down-box

You can assume that all sliders of the bank have the same aspect. In this case, you can assign a negative number to *ityetable* without worrying about previously defining any table. Negative numbers have the same meaning of the corresponding positive identifiers with the difference that the same aspect is assigned to all sliders. You can also assign a random aspect to each slider by setting *ityetable* to a negative number lower than -7. The default value is zero (all sliders have the aspect of nice sliders, without having to provide a table).

You can add 20 to a value inside the table to make the slider "plastic", or subtract 20 if you want to set the value for all widgets without defining a table (e.g. -21 to set all sliders types to Plastic Fill slider).

*iwidth* (optional) -- width of the rectangular area containing all sliders of the bank, excluding text labels, that are placed to the left of that area.

*iheight* (optional) -- height of the rectangular area containing all sliders of the bank, excluding text labels, that are placed to the left of that area.

*ix* (optional) -- horizontal position of the upper left corner of the rectangular area containing all sliders belonging to the bank. You have to leave enough space, at the left of that rectangle, in order to make sure labels of sliders to be visible. This is because the labels themselves are external to the rectangular area.

*iy* (optional) -- vertical position of the upper left corner of the rectangular area containing all sliders belonging to the bank. You have to leave enough space, at the left of that rectangle, in order to make sure labels of sliders to be visible. This is because the labels themselves are external to the rectangular area.

## Performance

There are no k-rate arguments, even if cells of the output table (or the zak space) are updated at k-rate.

*FLslidBnk* is a widget containing a bank of horizontal sliders. Any number of sliders can be placed into the bank (*inumsliders* argument). The output of all sliders is stored into a previously allocated table or into the zak space (*ioutable* argument). It is possible to determine the first location of the table (or of the zak space) in which to store the output of the first slider by means of *istart\_index* argument.

Each slider can have an individual label that is placed to the left of it. Labels are defined by the "*names*" argument. The output range of each slider can be individually set by means of an external table (*imin-maxtable* argument). The curve response of each slider can be set individually, by means of a list of identifiers placed in a table (*ixptable* argument). It is possible to define the aspect of each slider independently or to make all sliders have the same aspect (*ityetable* argument).

The *iwidth*, *iheight*, *ix*, and *iy* arguments determine width, height, horizontal and vertical position of the rectangular area containing sliders. Notice that the label of each slider is placed to the left of them and is not included in the rectangular area containing sliders. So the user should leave enough space to the left of the bank by assigning a proper *ix* value in order to leave labels visible.



## IMPORTANT!

Notice that the tables used by *FLslidBnk* must be created with the *ftgen* opcode and placed in the orchestra before the corresponding valuator. They can not placed in the score. This is because tables placed in the score are created later than the initialization of the opcodes placed in the header section of the orchestra.

## Examples

Here is an example of the *FLslidBnk* opcode. It uses the file *FLslidBnk.csd* [examples/FLslidBnk.csd].

### Example 305. Example of the *FLslidBnk* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc      -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o FLslidBnk.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 441
ksmps = 100
nchnls = 1

gitypetable ftgen 0, 0, 8, -2, 1, 1, 3, 3, 5, 5, 7, 7
giouttable ftgen 0, 0, 8, -2, 0, 0.2, 0.3, 0.4, 0.5, 0.6, 0.8, 1

FLpanel "Slider Bank", 400, 380, 50, 50
;Number of sliders
inum = 8
; Table to store output
iouttable = giouttable
; Width of the slider bank in pixels
iwidth = 350
; Height of the slider in pixels
iheight = 160
; Distance of the left edge of the slider
; from the left edge of the panel
ix = 30
; Distance of the top edge of the slider
; from the top edge of the panel
iy = 10
; Table containing fader types
itypetable = gitypetable
FLslidBnk "1@2@3@4@5@6@7@8", inum , iouttable , iwidth , iheight , ix \
, iy , itypetable
FLslidBnk "1@2@3@4@5@6@7@8", inum , iouttable , iwidth , iheight , ix \
, iy + 200 , -23
; End of panel contents
FLpanelEnd
; Run the widget thread!
FLrun

instr 1
```

```
;Dummy instrument
endin

</CsInstruments>
<CsScore>

; Instrument 1 will play a note for 1 hour.
i 1 0 3600
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*FLslider, FLslidBnk2, FLvslidBnk, FLvslidBnk2*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLslidBnk2

FLslidBnk2 — A FLTK widget containing a bank of horizontal sliders.

## Description

*FLslidBnk2* is a widget containing a bank of horizontal sliders.

## Syntax

```
FLslidBnk2 "names", inumsliders, ioutable, iconfigtable [,iwidth, iheight, ix, iy, istart_index]
```

```
FLslidBnk2 istring, inumsliders, ioutable, iconfigtable [,iwidth, iheight, ix, iy, istart_index]
```

## Initialization

*"names"* -- a double-quoted string containing the names of each slider. Each slider can have a different name. Separate each name with "@" character, for example: "frequency@amplitude@cutoff". It is possible to not provide any name by giving a single space " ". In this case, the opcode will automatically assign a progressive number as a label for each slider.

*istring* -- an integer associated with a string by *strset*, which is used like *"names"* above.

*inumsliders* -- the number of sliders.

*ioutable* (optional, default=0) -- number of a previously-allocated table in which to store output values of each slider. The user must be sure that table size is large enough to contain all output cells, otherwise a segfault will crash Csound. By assigning zero to this argument, the output will be directed to the zak space in the k-rate zone. In this case, the zak space must be previously allocated with the *zakinit* opcode and the user must be sure that the allocation size is big enough to cover all sliders. The default value is zero (i.e. store output in zak space).

*iconfigtable* -- in the *FLslidBnk2* and *FLvslidBnk2* opcodes, this table replaces *iminmaxtable*, *iexptable* and *istyletable*, all these parameters being placed into a single table. This table has to be filled with a group of 4 parameters for each slider in this way:

min1, max1, exp1, style1, min2, max2, exp2, style2, min3, max3, exp3, style3 etc.

for example using GEN02 you can type:

```
inum ftgen 1,0,256, -2, 0,1,0,1, 100, 5000, -1, 3, 50, 200, -1, 5,..... [etcetera]
```

In this example the first slider will be affected by the [0,1,0,1] parameters (the range will be 0 to 1, it will have linear response, and its aspect will be a fill slider), the second slider will be affected by the [100,5000,-1,3] parameters (the range is 100 to 5000, the response is exponential and the aspect is a normal slider), the third slider will be affected by the [50,200,-1,5] parameters (the range is 50 to 200, the behavior exponential, and the aspect is a nice slider), and so on.

*iwidth* (optional) -- width of the rectangular area containing all sliders of the bank, excluding text labels, that are placed to the left of that area.

*iheight* (optional) -- height of the rectangular area containing all sliders of the bank, excluding text labels, that are placed to the left of that area.

*ix* (optional) -- horizontal position of the upper left corner of the rectangular area containing all sliders belonging to the bank. You have to leave enough space, at the left of that rectangle, in order to make sure labels of sliders to be visible. This is because the labels themselves are external to the rectangular area.

*iy* (optional) -- vertical position of the upper left corner of the rectangular area containing all sliders belonging to the bank. You have to leave enough space, at the left of that rectangle, in order to make sure labels of sliders to be visible. This is because the labels themselves are external to the rectangular area.

*istart\_index* (optional, default=0) -- an integer number referring to a starting offset of output cell locations. It can be positive to allow multiple banks of sliders to output in the same table or in the zak space. The default value is zero (no offset).

## Performance

There are no k-rate arguments, even if cells of the output table (or the zak space) are updated at k-rate.

*FLslidBnk2* is a widget containing a bank of horizontal sliders. Any number of sliders can be placed into the bank (*inumsliders* argument). The output of all sliders is stored into a previously allocated table or into the zak space (*ioutable* argument). It is possible to determine the first location of the table (or of the zak space) in which to store the output of the first slider by means of *istart\_index* argument.

Each slider can have an individual label that is placed to the left of it. Labels are defined by the “*names*” argument. The output range of each slider can be individually set by means of the *min* and *max* values inside the *iconfigtable* table. The curve response of each slider can be set individually, by means of a list of identifiers placed in the *iconfigtable* table (*exp* argument). It is possible to define the aspect of each slider independently or to make all sliders have the same aspect (*style* argument in the *iconfigtable* table).

The *iwidth*, *iheight*, *ix*, and *iy* arguments determine width, height, horizontal and vertical position of the rectangular area containing sliders. Notice that the label of each slider is placed to the left of them and is not included in the rectangular area containing sliders. So the user should leave enough space to the left of the bank by assigning a proper *ix* value in order to leave labels visible.



### IMPORTANT!

Notice that the tables used by *FLslidBnk2* must be created with the *ftgen* opcode and placed in the orchestra before the corresponding valuator. They can not be placed in the score. This is because tables placed in the score are created later than the initialization of the opcodes placed in the header section of the orchestra.

## Examples

Here is an example of the *FLslidBnk2* opcode. It uses the file *FLslidBnk2.csd* [examples/*FLslidBnk2.csd*].

### Example 306. Example of the *FLslidBnk2* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc          -M0 ;;;RT audio I/O with MIDI in
</CsOptions>
<CsInstruments>
```

```

sr = 44100
ksmps = 100
nchnls = 2

;Example by Gabriel Maldonado

giElem init 8
giOutTab ftgen 1,0,128, 2, 0

;min1, max1, exp1, type1, min2, max2, exp2, type2, min3, max3, exp3, type3 etc.
giConfigTab ftgen 2,0,128,-2, .1, 1000, -1, 3, .1, 1000, -1, 3, .1, 1000, -1, 3,
.1, 5000, -1, 5, .1, 5000, -1, 5, .1, 5000, -1, 5,
giSine ftgen 3,0,256,10, 1

FLpanel "This Panel contains a Slider Bank",600,600
FLslidBnk2 "mod1@mod2@mod3&@freq1@freq2@freq3@freqPo", giElem, giOutTab, giConfigTab, 400, 500, 100
FLpanel_end

FLrun

instr 1

kmodindex1 init 0
kmodindex2 init 0
kmodindex3 init 0
kamp init 0
kfreq1 init 0
kfreq2 init 0
kfreq3 init 0
kfreq4 init 0

vtblk giOutTab, kmodindex1 , kmodindex2, kmodindex3, kamp, kfreq1, kfreq2 , kfreq3, kfreq4

amod1 oscili kmodindex1, kfreq1, giSine
amod2 oscili kmodindex2, kfreq2, giSine
amod3 oscili kmodindex3, kfreq3, giSine
aout oscili kamp, kfreq4+amod1+amod2+amod3, giSine

outs aout, aout
endin

</CsInstruments>
<CsScore>

i1 0 3600
f0 3600

</CsScore>
</CsoundSynthesizer>

```

## See Also

*FLslider, FLslidBnk, FLvslidBnk, FLvslidBnk2*

## Credits

Author: Gabriel Maldonado

New in version 5.06

# FLslidBnkGetHandle

FLslidBnkGetHandle — gets the handle of last slider bank created.

## Description

*FLslidBnkGetHandle* gets the handle of last slider bank created.

## Syntax

```
ihandle FLslidBnkGetHandle
```

## Initialization

*ihandle* - handle of the sliderBnk (to be used to set its values).

## Performance

There are no k-rate arguments, even if cells of the output table (or the zak space) are updated at k-rate.

*FLslidBnkGetHandle* gets the handle of last slider bank created. This opcode must follow corresponding *FLslidBnk* (or *FLvslidBnk*, *FLslidBnk2* and *FLvslidBnk2*) immediately, in order to get its handle.

See the entry for *FLslidBnk2Setk* to see an example of usage.

## See Also

*FLslider*, *FLslidBnk*, *FLslidBnk2*, *FLvslidBnk*, *FLvslidBnk2*, *FLslidBnk2Set*, *FLslidBnk2Setk*

## Credits

Author: Gabriel Maldonado

New in version 5.06

# FLslidBnkSet

FLslidBnkSet — modify the values of a slider bank.

## Description

*FLslidBnkSet* modifies the values of a slider bank according to an array of values stored in a table.

## Syntax

```
FLslidBnkSet ihandle, ifn [, istartIndex, istartSlid, inumSlid]
```

## Initialization

*ihandle* - handle of the sliderBnk (to be used to set its values).

*ifn* - number of a table containing an array of values to set each slider to.

*istartIndex* - (optional) starting index of the table element of to be evaluated firstly. Default value is zero

*istartSlid* - (optional) starting slider to be evaluated. Default 0, denoting the first slider.

*inumSlid* - (optional) number of sliders to be updated. Default 0, denoting all sliders.

## Performance

*FLslidBnkSet* modifies the values of a slider bank (created with *FLslidBnk* or with *FLvslidBnk*) according to an array of values stored into table *ifn*. It actually allows to update an *FLslidBnk* (or *FLvslidBnk*) bank of sliders (for instance, using the *slider8table* opcode) to a set of values located in a table. User has to set *ihandle* argument to the handle got from *FLslidBnkGetHandle* opcode. It works at init-rate only. It is possible to reset only a range of sliders, by using the optional arguments *istartIndex*, *istartSlid*, *inumSlid*

There is a k-rate version of this opcode called *FLslidBnkSetk*.

## See Also

*FLslider*, *FLslidBnkGetHandle*, *FLslidBnk*, *FLslidBnk2*, *FLvslidBnk*, *FLvslidBnk2* *FLslidBnk2Set*, *FLslidBnkSetk*

## Credits

Author: Gabriel Maldonado

New in version 5.06

# FLslidBnkSetk

FLslidBnkSetk — modify the values of a slider bank.

## Description

*FLslidBnkSetk* modifies the values of a slider bank according to an array of values stored in a table.

## Syntax

```
FLslidBnkSetk ktrig, ihandle, ifn [, istartIndex, istartSlid, inumSlid]
```

## Initialization

*ihandle* - handle of the sliderBnk (to be used to set its values).

*ifn* - number of a table containing an array of values to set each slider to.

*istartIndex* - (optional) starting index of the table element of to be evaluated firstly. Default value is zero

*istartSlid* - (optional) starting slider to be evaluated. Default 0, denoting the first slider.

*inumSlid* - (optional) number of sliders to be updated. Default 0, denoting all sliders.

## Performance

*ktrig* – the output of *FLslidBnkSetk* consists of a trigger that informs if sliders have to be updated or not. A non-zero value forces the slider to be updated.

*FLslidBnkSetk* is similar to *FLslidBnkSet* but allows k-rate to modify the values of *FLslidBnk* (*FLslidBnkSetk* can also be used with *FLvslidBnk*, obtaining identical result). It also allows the slider bank to be joined with MIDI. If you are using MIDI (for instance, when using the *slider8table* opcode), *FLslidBnkSetk* changes the values of *FLslidBnk* bank of sliders to a set of values located in a table. This opcode is actually able to serve as a MIDI bridge to the *FLslidBnk* widget when used together with the *sliderXXtable* set of opcodes (see *slider8table* entry for more information). Notice, that, when you want to use table indexing as a curve response, it is not possible to do it directly in the *iconfigtable* configuration of *FLslidBnk2*, when you intend to use the *FLslidBnkSetk* opcode. In fact, corresponding *inputTable* element of *FLslidBnkSetk* must be set in linear mode and respect the 0 to 1 range. Even the corresponding elements of *sliderXXtable* must be set in linear mode and in the normalized range. You can do table indexing later, by using the *tab* and *tb* opcodes, and rescaling output according to max and min values. By the other hand, it is possible to use linear and exponential curve response directly, by setting the actual min-max range and flag both in the *iconfigtable* of corresponding *FLslidBnk2* and in *sliderXXtable*.

*FLslidBnkSetk* the k-rate version of *FLslidBnk2Set*.

## See Also

*FLslider*, *FLslidBnkGetHandle*, *FLslidBnk*, *FLslidBnk2*, *FLvslidBnk*, *FLvslidBnk2* *FLslidBnkSet*, *FLslidBnk2Set*, *slider8table*

## Credits

Author: Gabriel Maldonado

New in version 5.06

# FLslidBnk2Set

FLslidBnk2Set — modify the values of a slider bank.

## Description

*FLslidBnk2Set* modifies the values of a slider bank according to an array of values stored in a table.

## Syntax

```
FLslidBnk2Set ihandle, ifn [, istartIndex, istartSlid, inumSlid]
```

## Initialization

*ihandle* - handle of the sliderBnk (to be used to set its values).

*ifn* - number of a table containing an array of values to set each slider to.

*istartIndex* - (optional) starting index of the table element of to be evaluated firstly. Default value is zero

*istartSlid* - (optional) starting slider to be evaluated. Default 0, denoting the first slider.

*inumSlid* - (optional) number of sliders to be updated. Default 0, denoting all sliders.

## Performance

*FLslidBnk2Set* modifies the values of a slider bank (created with *FLslidBnk2* or with *FLvslidBnk2*) according to an array of values stored into table *ifn*. It actually allows to update an *FLslidBnk2* (or *FLvslidBnk2*) bank of sliders (for instance, using the *slider8table* opcode) to a set of values located in a table. User has to set *ihandle* argument to the handle got from *FLslidBnkGetHandle* opcode. It works at init-rate only. It is possible to reset only a range of sliders, by using the optional arguments *istartIndex*, *istartSlid*, *inumSlid*

*FLslidBnk2Set* is identical to *FLslidBnkSet*, but works on *FLslidBnk2* and *FLvslidBnk2* instead of *FLslidBnk* and *FLvslidBnk*.

There is a k-rate version of this opcode called *FLslidBnk2Setk*.

## See Also

*FLslider*, *FLslidBnkGetHandle*, *FLslidBnk*, *FLslidBnk2*, *FLvslidBnk*, *FLvslidBnk2* *FLslidBnkSet*, *FLslidBnk2Setk*

## Credits

Author: Gabriel Maldonado

New in version 5.06

# FLslidBnk2Setk

FLslidBnk2Setk — modify the values of a slider bank.

## Description

*FLslidBnk2Setk* modifies the values of a slider bank according to an array of values stored in a table.

## Syntax

```
FLslidBnk2Setk ktrig, ihandle, ifn [, istartIndex, istartSlid, inumSlid]
```

## Initialization

*ihandle* - handle of the sliderBnk (to be used to set its values).

*ifn* - number of a table containing an array of values to set each slider to.

*istartIndex* - (optional) starting index of the table element of to be evaluated firstly. Default value is zero

*istartSlid* - (optional) starting slider to be evaluated. Default 0, denoting the first slider.

*inumSlid* - (optional) number of sliders to be updated. Default 0, denoting all sliders.

## Performance

*ktrig* – the output of *FLslidBnk2Setk* consists of a trigger that informs if sliders have to be updated or not. A non-zero value forces the slider to be updated.

*FLslidBnk2Setk* is similar to *FLslidBnkSet* but allows k-rate to modify the values of *FLslidBnk2* (*FLslidBnk2Setk* can also be used with *FLvslidBnk2*, obtaining identical result). It also allows the slider bank to be joined with MIDI. If you are using MIDI (for instance, when using the *slider8table* opcode), *FLslidBnk2Setk* changes the values of *FLslidBnk2* bank of sliders to a set of values located in a table. This opcode is actually able to serve as a MIDI bridge to the *FLslidBnk2* widget when used together with the *sliderXXtable* set of opcodes (see *slider8table* entry for more information). Notice, that, when you want to use table indexing as a curve response, it is not possible to do it directly in the *iconfigtable* configuration of *FLslidBnk2*, when you intend to use the *FLslidBnk2Setk* opcode. In fact, corresponding *inputTable* element of *FLslidBnk2Setk* must be set in linear mode and respect the 0 to 1 range. Even the corresponding elements of *sliderXXtable* must be set in linear mode and in the normalized range. You can do table indexing later, by using the *tab* and *tb* opcodes, and rescaling output according to max and min values. By the other hand, it is possible to use linear and exponential curve response directly, by setting the actual min-max range and flag both in the *iconfigtable* of corresponding *FLslidBnk2* and in *sliderXXtable*.

*FLslidBnk2Setk* the k-rate version of *FLslidBnk2Set*.

## Examples

Here is an example of the *FLslidBnk2Setk* opcode. It uses the file *FLslidBnk2Setk.csd* [examples/FLslidBnk2Setk.csd].

**Example 307. Example of the FLslidBnk2Setk opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Audio out   Audio in   No messages
-odac        -iadc      -d          ;;RT audio I/O
</CsOptions>
<CsInstruments>

sr          = 44100
ksmps      = 10
nchnls     = 2

;Example by Gabriel Maldonado 2007
giElem init 8
giOutTab ftgen 1,0,128, 2, 0
giSine ftgen 3,0,256,10, 1
giOutTab2 ftgen 4,0,128, 2, 0
itab ftgen 29, 0, 129, 5, .002, 128, 1 ;** exponential ascending curve for slider mapping
giExpTab ftgen 30, 0, 129, -24, itab, 0, 1 ;** rescaled curve for slider mapping
giConfigTab ftgen 2,0,128,-2, 1, 500, -1, 13, \
          1, 500, -1, 13, \
          1, 500, -1, 13, \
          1, 5000, -1, 13, \
\
          1, 1000, -1, 5, \
          1, 1000, -1, 5, \
          1, 1000, -1, 5, \
          1, 5000, -1, 5

FLpanel "Multiple FM",600,600
FLslidBnk2 "mod1@mod2@mod3@amp@freq1@freq2@freq3@freqPo", giElem, giOutTab2, giConfigTab, 400, 500, 1000
giHandle FLslidBnkGetHandle
FLpanel_end

FLrun
instr 1
ktrig slider8table 1, giOutTab, 0, \ ; ctl min max init func
27, 1, 500, 3, -1, \ ;1 repeat rate
28, 1, 500, 4, -1, \ ;2 random freq. amount
29, 1, 500, 1, -1, \ ;3 random amp. amount
30, 1, 5000, 1, -1, \ ;4 number of concurrent loop points
\
31, 1, 1000, 1, -1, \ ;5 kloop1
32, 1, 1000, 1, -1, \ ;6 kloop2
33, 1, 1000, 1, -1, \ ;7 kloop3
34, 1, 1000, 1, -1 ;8 kloop4
kmodindex1 init 0
kmodindex2 init 0
kmodindex3 init 0
kamp init 0
kfreq1 init 0
kfreq2 init 0
kfreq3 init 0
kfreq4 init 0
vtablelk giOutTab2, kmodindex1, kmodindex2, kmodindex3, kamp, kfreq1, kfreq2, kfreq3, kfreq4
; *kflag, *ihandle, *ifn, *startInd, *startSlid, *numSlid;
FLslidBnk2Setk ktrig, giHandle, giOutTab, 0, 0, giElem
printk2 kmodindex1
printk2 kmodindex2,10
printk2 kmodindex3,20
printk2 kamp,30
amod1 oscili kmodindex1, kfreq1, giSine

```

```
amod2 oscili kmodindex2, kfreq2, giSine
amod3 oscili kmodindex3, kfreq3, giSine
aout oscili kamp,          kfreq4+amod1+amod2+amod3, giSine
outs aout, aout
endin
</CsInstruments>

<CsScore>
i1 0 3600
</CsScore>

</CsoundSynthesizer>
```

## See Also

*FLslider, FLslidBnkGetHandle, FLslidBnk, FLslidBnk2, FLvslidBnk, FLvslidBnk2 FLslidBnkSet, FLslidBnk2Set, slider8table*

## Credits

Author: Gabriel Maldonado

New in version 5.06

# FLslider

FLslider — Puts a slider into the corresponding FLTK container.

## Description

*FLslider* puts a slider into the corresponding container.

## Syntax

```
kout, ihandle FLslider "label", imin, imax, iexp, itype, idisp, iwidth, \  
    iheight, ix, iy
```

## Initialization

*ihandle* -- a handle value (an integer number) that unequivocally references a corresponding widget. This is used by other opcodes that modify a widget's properties (see *Modifying FLTK Widget Appearance*). It is automatically output by *FLslider* and must not be set by the user label. (The user label is a double-quoted string containing some user-provided text placed near the widget.)

*label* -- a double-quoted string containing some user-provided text, placed near the corresponding widget.

*imin* -- minimum value of output range (corresponds to the left value for horizontal sliders, and the top value for vertical sliders).

*imax* -- maximum value of output range (corresponds to the right value for horizontal sliders, and the bottom value for vertical sliders).

The *imin* argument may be greater than *imax* argument. This has the effect of “reversing” the object so the larger values are in the opposite direction. This also switches which end of the filled sliders is filled.

*iexp* -- an integer number denoting the behaviour of valuator:

- 0 = valuator output is linear
- -1 = valuator output is exponential

All other positive numbers for *iexp* indicate the number of an existing table that is used for indexing. Linear interpolation is provided in table indexing. A negative table number suppresses interpolation.



### IMPORTANT!

Notice that the tables used by valuator must be created with the *ftgen* opcode and placed in the orchestra before the corresponding valuator. They can not placed in the score. This is because tables placed in the score are created later than the initialization of the opcodes placed in the header section of the orchestra.

*itype* -- an integer number denoting the appearance of the valuator.

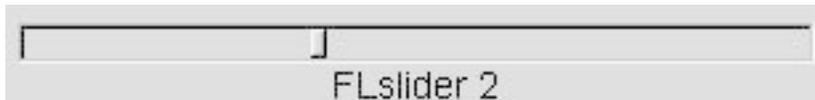
The *itype* argument can be set to the following values:

- 1 - shows a horizontal fill slider

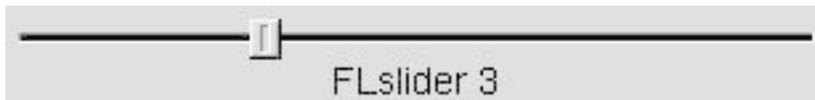
- 2 - a vertical fill slider
- 3 - a horizontal engraved slider
- 4 - a vertical engraved slider
- 5 - a horizontal nice slider
- 6 - a vertical nice slider
- 7 - a horizontal up-box nice slider
- 8 - a vertical up-box nice slider



FLslider - a horizontal fill slider (*itype*=1).



FLslider - a horizontal engraved slider (*itype*=3).



FLslider - a horizontal nice slider (*itype*=5).

You can also create "plastic" looking sliders by adding 20 to *itype*.

*idisp* -- a handle value that was output from a previous instance of the *FLvalue* opcode to display the current value of the current valuator in the *FLvalue* widget itself. If the user doesn't want to use this feature that displays current values, it must be set to a negative number by the user.

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

## Performance

*kout* -- output value

FLsliders are created with the minimum value by default in the left/at the top. If you want to reverse the slider, reverse the values. See the example below.

## Examples

Here is an example of the FLslider opcode. It uses the file *FLslider.csd* [examples/FLslider.csd].

**Example 308. Example of the FLslider opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d          ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o FLslider.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; A sine with oscillator with flslider controlled frequency
sr = 44100
kr = 441
ksmps = 100
nchnls = 1

FLpanel "Frequency Slider", 900, 400, 50, 50
; Minimum value output by the slider
imin = 200
; Maximum value output by the slider
imax = 5000
; Logarithmic type slider selected
iexp = -1
; Slider graphic type (5='nice' slider)
itype = 5
; Display handle (-1=not used)
idisp = -1
; Width of the slider in pixels
iwidth = 750
; Height of the slider in pixels
iheight = 30
; Distance of the left edge of the slider
; from the left edge of the panel
ix = 125
; Distance of the top edge of the slider
; from the top edge of the panel
iy = 50

gkfreq, ihandle FLslider "Frequency", imin, imax, iexp, itype, idisp, iwidth, iheight, ix, iy
; End of panel contents
FLpanelEnd
; Run the widget thread!
FLrun

;Set the widget's initial value
FLsetVal_i 300, ihandle

instr 1
iamp = 15000
ifn = 1
kfreq portk gkfreq, 0.005 ;Smooth gkfreq to avoid zipper noise
asig oscili iamp, kfreq, ifn
out asig
endin

</CsInstruments>
<CsScore>

; Function table that defines a single cycle
; of a sine wave.

```

```
f 1 0 1024 10 1
; Instrument 1 will play a note for 1 hour.
i 1 0 3600
e
```

```
</CsScore>
</CsoundSynthesizer>
```

Here is another example of the FLslider opcode, showing the slider types and other options. It also shows the usage of FLvalue to display a widget's contents. It uses the file *FLslider-2.csd* [examples/FLslider-2.csd].

### Example 309. More complex example of the FLslider opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in   No messages
-odac         -iadc      -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o FLslider-2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 441
ksmps = 100
nchnls = 1

;By Andres Cabrera 2007

FLpanel "Slider Types", 410, 260, 50, 50
; Distance of the left edge of the slider
; from the left edge of the panel
ix = 10
; Distance of the top edge of the slider
; from the top edge of the panel
iy = 10
; Create boxes to display widget values
givalue1 FLvalue "1", 60, 20, ix + 330, iy
givalue3 FLvalue "3", 60, 20, ix + 330, iy + 40
givalue5 FLvalue "5", 60, 20, ix + 330, iy + 80

givalue2 FLvalue "2", 60, 20, ix + 60, iy + 140
givalue4 FLvalue "4", 60, 20, ix + 195, iy + 140
givalue6 FLvalue "6", 60, 20, ix + 320, iy + 140

;Horizontal sliders
gkdummy1, gihandle1 FLslider "Type 1", 200, 5000, -1, 1, givalue1, 320, 20, ix, iy
gkdummy3, gihandle3 FLslider "Type 3", 0, 15000, 0, 3, givalue3, 320, 20, ix, iy + 40
; Reversed slider
gkdummy5, gihandle5 FLslider "Type 5", 1, 0, 0, 5, givalue5, 320, 20, ix, iy + 80

;Vertical sliders
gkdummy2, gihandle2 FLslider "Type 2", 0, 1, 0, 2, givalue2, 20, 100, ix+ 30 , iy + 120
; Reversed slider
gkdummy4, gihandle4 FLslider "Type 4", 1, 0, 0, 4, givalue4, 20, 100, ix + 165 , iy + 120
gkdummy6, gihandle6 FLslider "Type 6", 0, 1, 0, 6, givalue6, 20, 100, ix + 290 , iy + 120
FLpanelEnd

FLpanel "Plastic Slider Types", 410, 300, 150, 150
; Distance of the left edge of the slider
; from the left edge of the panel
```

```

ix = 10
; Distance of the top edge of the slider
; from the top edge of the panel
iy = 10
; Create boxes to display widget values
givalue21 FLvalue "21", 60, 20, ix + 330, iy
givalue23 FLvalue "23", 60, 20, ix + 330, iy + 40
givalue25 FLvalue "25", 60, 20, ix + 330, iy + 80

givalue22 FLvalue "22", 60, 20, ix + 60, iy + 140
givalue24 FLvalue "24", 60, 20, ix + 195, iy + 140
givalue26 FLvalue "26", 60, 20, ix + 320, iy + 140

;Horizontal sliders
gkdummy21, gihandle21 FLslider "Type 21", 200, 5000, -1, 21, givalue21, 320, 20, ix, iy
gkdummy23, gihandle23 FLslider "Type 23", 0, 15000, 0, 23, givalue23, 320, 20, ix, iy + 40
; Reversed slider
gkdummy25, gihandle25 FLslider "Type 25", 1, 0, 0, 25, givalue25, 320, 20, ix, iy + 80

;Vertical sliders
gkdummy22, gihandle22 FLslider "Type 22", 0, 1, 0, 22, givalue22, 20, 100, ix+ 30 , iy + 120
; Reversed slider
gkdummy24, gihandle24 FLslider "Type 24", 1, 0, 0, 24, givalue24, 20, 100, ix + 165 , iy + 120
gkdummy26, gihandle26 FLslider "Type 26", 0, 1, 0, 26, givalue26, 20, 100, ix + 290 , iy + 120
;Button to add color to the sliders
gkcolors, ihdummy FLbutton "Color", 1, 0, 21, 150, 30, 30, 260, 0, 10, 0, 1

FLpanelEnd
FLrun

;Set some widget's initial value
FLsetVal_i 500, gihandle1
FLsetVal_i 1000, gihandle3

instr 10
; Set the color of widgets
FLsetColor 200, 230, 0, gihandle1
FLsetColor 0, 123, 100, gihandle2
FLsetColor 180, 23, 12, gihandle3
FLsetColor 10, 230, 0, gihandle4
FLsetColor 0, 0, 0, gihandle5
FLsetColor 0, 0, 0, gihandle6

FLsetColor 200, 230, 0, givalue1
FLsetColor 0, 123, 100, givalue2
FLsetColor 180, 23, 12, givalue3
FLsetColor 10, 230, 0, givalue4
FLsetColor 255, 255, 255, givalue5
FLsetColor 255, 255, 255, givalue6

FLsetColor2 20, 23, 100, gihandle1
FLsetColor2 200,0 ,123 , gihandle2
FLsetColor2 180, 180, 100, gihandle3
FLsetColor2 180, 23, 12, gihandle4
FLsetColor2 180, 180, 100, gihandle5
FLsetColor2 180, 23, 12, gihandle6

FLsetColor 200, 230, 0, gihandle21
FLsetColor 0, 123, 100, gihandle22
FLsetColor 180, 23, 12, gihandle23
FLsetColor 10, 230, 0, gihandle24
FLsetColor 0, 0, 0, gihandle25
FLsetColor 0, 0, 0, gihandle26

FLsetColor 200, 230, 0, givalue21
FLsetColor 0, 123, 100, givalue22

```

```
FLsetColor 180, 23, 12, givalue23
FLsetColor 10, 230, 0, givalue24
FLsetColor 255, 255, 255, givalue25
FLsetColor 255, 255, 255, givalue26

FLsetColor2 20, 23, 100, gihandle21
FLsetColor2 200,0 ,123 , gihandle22
FLsetColor2 180, 180, 100, gihandle23
FLsetColor2 180, 23, 12, gihandle24
FLsetColor2 180, 180, 100, gihandle25
FLsetColor2 180, 23, 12, gihandle26

; Slider values must be updated for colors to change
FLsetVal_i 250, gihandle1
FLsetVal_i 0.5, gihandle2
FLsetVal_i 0, gihandle3
FLsetVal_i 0, gihandle4
FLsetVal_i 0, gihandle5
FLsetVal_i 0.5, gihandle6
FLsetVal_i 250, gihandle21
FLsetVal_i 0.5, gihandle22
FLsetVal_i 500, gihandle23
FLsetVal_i 0, gihandle24
FLsetVal_i 0, gihandle25
FLsetVal_i 0.5, gihandle26

endin

</CsInstruments>
<CsScore>
f 0 3600 ;Dumy table to make csound wait for realtime events

e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*FLcount, FLjoy, FLknob, FLroller, FLslidBnk, FLtext*

## Credits

Author: Gabriel Maldonado

New in version 4.22

February 2004. Thanks to a note from Dave Phillips, deleted the extraneous istep parameter.

Example written by Iain McCurdy, edited by Kevin Conder.

# FLtabs

FLtabs — Creates a tabbed FLTK interface.

## Description

*FLtabs* is a “file card tabs” interface that is useful to display several areas containing widgets in the same windows, alternatively. It must be used together with *FLgroup*, another container that groups child widgets.

## Syntax

```
FLtabs iwidth, iheight, ix, iy
```

## Initialization

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window. Expressed in pixels.

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window. Expressed in pixels.

## Performance

Containers are useful to format the graphic appearance of the widgets. The most important container is *FLpanel*, that actually creates a window. It can be filled with other containers and/or valuators or other kinds of widgets.

There are no k-rate arguments in containers.

*FLtabs* is a “file card tabs” interface that is useful to display several alternate areas containing widgets in the same window.



FLtabs.

It must be used together with *FLgroup*, another FLTK container opcode that groups child widgets.

## Examples

The following example code:

```
FLpanel "Panel1", 450, 550, 100, 100
FLscroll 450, 550, 0, 0
```

```

FLtabs 400, 550, 5, 5

    FLgroup "sliders", 380, 500, 10, 40, 1
gk1, ihs FLslider "FLslider 1", 500, 1000, 2 ,1, -1, 300,15, 20,50
gk2, ihs FLslider "FLslider 2", 300, 5000, 2 ,3, -1, 300,15, 20,100
gk3, ihs FLslider "FLslider 3", 350, 1000, 2 ,5, -1, 300,15, 20,150
gk4, ihs FLslider "FLslider 4", 250, 5000, 1 ,11, -1, 300,30, 20,200
gk5, ihs FLslider "FLslider 5", 220, 8000, 2 ,1, -1, 300,15, 20,250
gk6, ihs FLslider "FLslider 6", 1, 5000, 1 ,13, -1, 300,15, 20,300
gk7, ihs FLslider "FLslider 7", 870, 5000, 1 ,15, -1, 300,30, 20,350
gk8, ihs FLslider "FLslider 8", 20, 20000, 2 ,6, -1, 30,400, 350,50
    FLgroupEnd

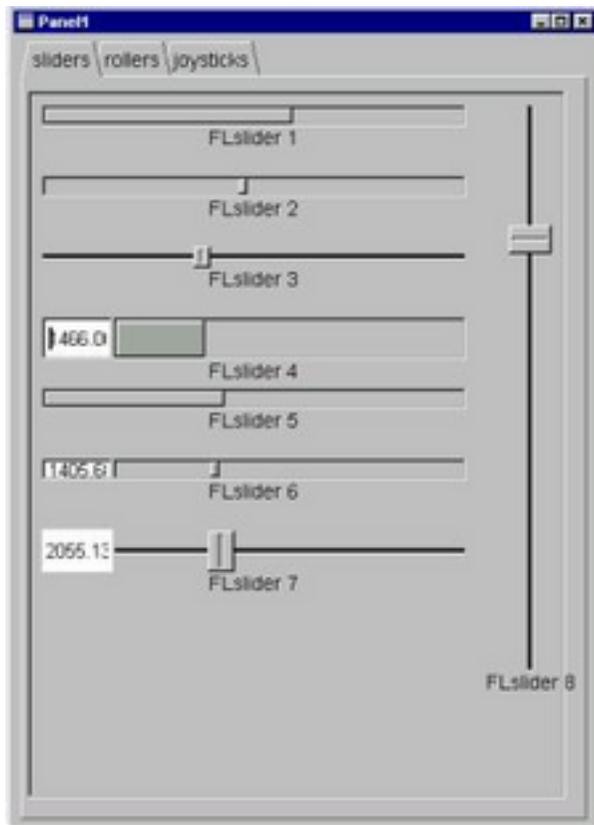
    FLgroup "rollers", 380, 500, 10, 30, 2
gk1, ihr FLroller "FLroller 1", 50, 1000, .1,2 ,1 , -1, 200,22, 20,50
gk2, ihr FLroller "FLroller 2", 80, 5000, .1,2 ,1 , -1, 200,22, 20,100
gk3, ihr FLroller "FLroller 3", 50, 1000, .1,2 ,1 , -1, 200,22, 20,150
gk4, ihr FLroller "FLroller 4", 80, 5000, .1,2 ,1 , -1, 200,22, 20,200
gk5, ihr FLroller "FLroller 5", 50, 1000, .1,2 ,1 , -1, 200,22, 20,250
gk6, ihr FLroller "FLroller 6", 80, 5000, .1,2 ,1 , -1, 200,22, 20,300
gk7, ihr FLroller "FLroller 7", 50, 5000, .1,1 ,2 , -1, 30,300, 280,50
    FLgroupEnd

    FLgroup "joysticks", 380, 500, 10, 40, 3
gk1, gk2, ihj1, ihj2 FLjoy "FLjoy", 50, 18000, 50, 18000, 2, 2, -1, -1, 300, 300, 30, 60
    FLgroupEnd

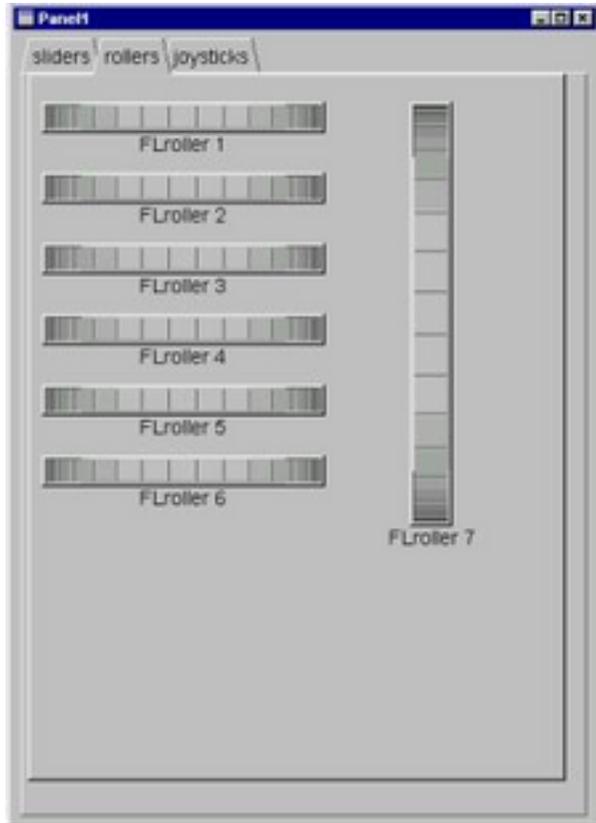
FLtabsEnd
FLscrollEnd
FLpanelEnd

```

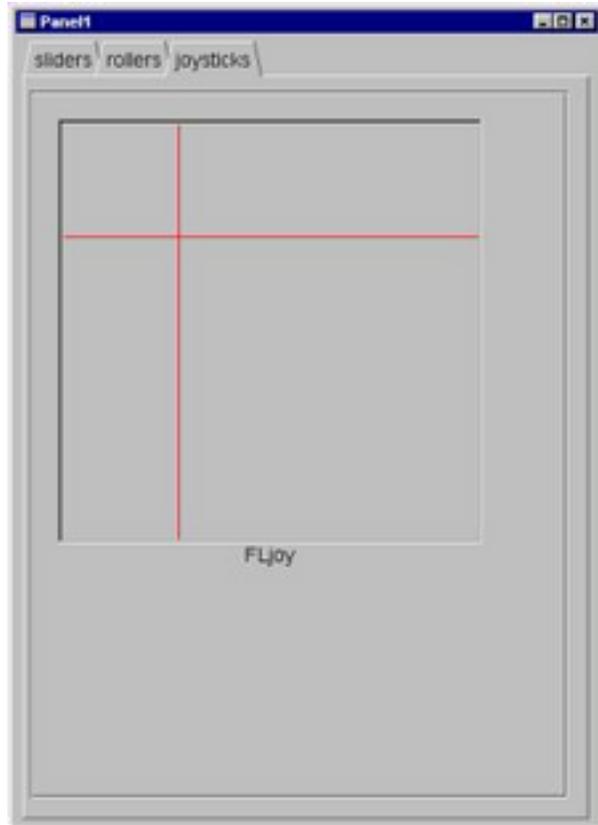
...will produce the following result:



FLtabs example, sliders tab.



FLtabs example, rollers tab.



FLtabs example, joysticks tab.  
(Each picture shows a different tab selection inside the same window.)

## Examples

Here is an example of the FLtabs opcode. It uses the file *FLtabs.csd* [examples/FLtabs.csd].

### Example 310. Example of the FLtabs opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d          ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o FLtabs.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; A single oscillator with frequency, amplitude and
; panning controls on separate file tab cards
sr = 44100
kr = 441
ksmps = 100
nchnls = 2
```

```

FLpanel "Tabs", 300, 350, 100, 100
itabswidth = 280
itabsheight = 330
ix = 5
iy = 5
FLtabs itabswidth,itabsheight, ix,iy

    itab1width = 280
    itab1height = 300
    itab1x = 10
    itab1y = 40
FLgroup "Tab 1", itab1width, itab1height, itab1x, itab1y
    gkfreq, i1 FLknob "Frequency", 200, 5000, -1, 1, -1, 70, 70, 130
    FLsetVal_i 400, i1
FLgroupEnd

    itab2width = 280
    itab2height = 300
    itab2x = 10
    itab2y = 40
FLgroup "Tab 2", itab2width, itab2height, itab2x, itab2y
    gkamp, i2 FLknob "Amplitude", 0, 15000, 0, 1, -1, 70, 70, 130
    FLsetVal_i 15000, i2
FLgroupEnd

    itab3width = 280
    itab3height = 300
    itab3x = 10
    itab3y = 40
FLgroup "Tab 3", itab3width, itab3height, itab3x, itab3y
    gkpan, i3 FLknob "Pan position", 0, 1, 0, 1, -1, 70, 70, 130
    FLsetVal_i 0.5, i3
FLgroupEnd
FLtabsEnd
FLpanelEnd
; Run the widget thread!
FLrun

instr 1
    ifn = 1
    asig oscili gkamp, gkfreq, ifn
    outs asig*(1-gkpan), asig*gkpan
endin

</CsInstruments>
<CsScore>

; Function table that defines a single cycle
; of a sine wave.
f 1 0 1024 10 1

; Instrument 1 will play a note for 1 hour.
i 1 0 3600
e

</CsScore>
</CsoundSynthesizer>

```

## See Also

*FLgroup, FLgroupEnd, FLpack, FLpackEnd, FLpanel, FLpanelEnd, FLscroll, FLscrollEnd, FLtabsEnd*

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.

# FLtabsEnd

FLtabsEnd — Marks the end of a tabbed FLTK interface.

## Description

Marks the end of a tabbed FLTK interface.

## Syntax

`FLtabsEnd`

## Performance

Containers are useful to format the graphic appearance of the widgets. The most important container is *FLpanel*, that actually creates a window. It can be filled with other containers and/or valuators or other kinds of widgets.

There are no k-rate arguments in containers.

## See Also

*FLgroup*, *FLgroupEnd*, *FLpack*, *FLpackEnd*, *FLpanel*, *FLpanelEnd*, *FLscroll*, *FLscrollEnd*, *FLtabs*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLtabs\_end

FLtabs\_end — Marks the end of a tabbed FLTK interface.

## Description

Marks the end of a tabbed FLTK interface. This is another name for **FLtabsEnd** provided for compatibility.  
See *FLtabsEnd*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLtext

FLtext — A FLTK widget opcode that creates a textbox.

## Description

*FLtext* allows the user to modify a parameter value by directly typing it into a text field.

## Syntax

```
kout, ihandle FLtext "label", imin, imax, istep, itype, iwidth, \  
    iheight, ix, iy
```

## Initialization

*ihandle* -- a handle value (an integer number) that unequivocally references a corresponding widget. This is used by other opcodes that modify a widget's properties (see *Modifying FLTK Widget Appearance*). It is automatically output by *FLtext* and must not be set by the user label. (The user label is a double-quoted string containing some user-provided text placed near the widget.)

*"label"* -- a double-quoted string containing some user-provided text, placed near corresponding widget.

*imin* -- minimum value of output range.

*imax* -- maximum value of output range.

*istep* -- a floating-point number indicating the increment of valuator value corresponding to the mouse dragging. The *istep* argument allows the user to arbitrarily slow mouse motion, enabling arbitrary precision.

*itype* -- an integer number denoting the appearance of the valuator.

The *itype* argument can be set to the following values:

- 1 - normal behaviour
- 2 - dragging operation is suppressed, instead it will appear two arrow buttons. A mouse-click on one of these buttons can increase/decrease the output value.
- 3 - text editing is suppressed, only mouse dragging modifies the output value.

*iwidth* -- width of widget.

*iheight* -- height of widget.

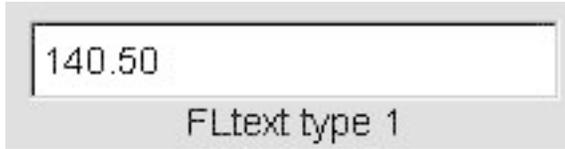
*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

## Performance

*kout* -- output value

*FLtext* allows the user to modify a parameter value by directly typing it into a text field:



*FLtext*.

Its value can also be modified by clicking on it and dragging the mouse horizontally. The *istep* argument allows the user to arbitrarily set the response on mouse dragging.

## Examples

Here is an example of the *FLtext* opcode. It uses the file *FLtext.csd* [examples/*FLtext.csd*].

### Example 311. Example of the *FLtext* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o FLtext.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; A sine with oscillator with fltext box controlled
; frequency either click and drag or double click and
; type to change frequency value
sr = 44100
kr = 441
ksmps = 100
nchnls = 1

FLpanel "Frequency Text Box", 270, 600, 50, 50
; Minimum value output by the text box
imin = 200
; Maximum value output by the text box
imax = 5000
; Step size
istep = 1
; Text box graphic type
itype = 1
; Width of the text box in pixels
iwidth = 70
; Height of the text box in pixels
iheight = 30
; Distance of the left edge of the text box
; from the left edge of the panel
ix = 100
; Distance of the top edge of the text box
; from the top edge of the panel
iy = 300

gkfreq,ihandle FLtext "Enter the frequency", imin, imax, istep, itype, iwidth, iheight, ix, iy
; End of panel contents
FLpanelEnd
; Run the widget thread!
```

```
FLrun

instr 1
  iamp = 15000
  ifn = 1
  asig oscili iamp, gkfreq, ifn
  out asig
endin

</CsInstruments>
<CsScore>

; Function table that defines a single cycle
; of a sine wave.
f 1 0 1024 10 1

; Instrument 1 will play a note for 1 hour.
i 1 0 3600
e

</CsScore>
</CsSoundSynthesizer>
```

## See Also

*FLcount, FLjoy, FLknob, FLroller, FLslider*

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.

# FLupdate

FLupdate — Same as the FLrun opcode.

## Description

Same as the *FLrun* opcode.

## Syntax

`FLupdate`

# fluidAllOut

fluidAllOut — Collects all audio from all Fluidsynth engines in a performance

## Syntax

```
aleft, aright fluidAllOut
```

## Description

Collects all audio from all Fluidsynth engines in a performance

## Performance

*aleft* -- Left channel audio output.

*aright* -- Right channel audio output.

Invoke fluidAllOut in an instrument definition numbered higher than any fluidcontrol instrument definitions. All SoundFonts send their audio output to this one opcode. Send a note with an indefinite duration to this instrument to turn the SoundFonts on for as long as required.

In this implementation, SoundFont effects such as chorus or reverb are used if and only if they are defaults for the preset. There is no means of turning such effects on or off, or of changing their parameters, from Csound.

## Examples

Here is an example of the fluidAllOut opcodes. It uses the file *fluidAllOut.csd* [examples/fluidAllOut.csd].

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -MO    ;;realtime audio out and realtime midi in
;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
;-o fluidAllOut.wav -W    ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giengine1 fluidEngine
isfnum1 fluidLoad "sf_GMbank.sf2", giengine1, 1
fluidProgramSelect giengine1, 1, isfnum1, 0, 0

giengine2 fluidEngine
; soundfont path to manual/examples
isfnum2 fluidLoad "22Bassoon.sf2", giengine2, 1
fluidProgramSelect giengine2, 1, isfnum2, 0, 70

instr 1
```

```
        mididefault 60, p3
        midinoteonkey p4, p5
ikey init p4
ivel init p5
    fluidNote giengine1, 1, ikey, ivel

    endin

instr 2

        mididefault 60, p3
        midinoteonkey p4, p5
ikey init p4
ivel init p5
    fluidNote giengine2, 1, ikey, ivel

    endin

instr 100

imvol init 7 ;amplify a bit
asigl, asigr fluidAllOut
    outs asigl*imvol, asigr*imvol

    endin
</CsInstruments>
<CsScore>

i 1 0 2 60 127 ;play one note on instr 1
i 2 2 2 60 127 ;play another note on instr 2 and...
i 100 0 60      ;play virtual midi keyboard
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*fluidEngine, fluidNote, fluidLoad*

More information on soundfonts in the Floss Manuals: [http://en.flossmanuals.net/csound/ch048\\_d-reading-midi-files](http://en.flossmanuals.net/csound/ch048_d-reading-midi-files)

Other information on soundfonts on Wikipedia: <http://en.wikipedia.org/wiki/Soundfont>

## Credits

Opcode by Michael Gogins (gogins at pipeline dot com). Thanks to Peter Hanappe for Fluidsynth, and to Steven Yi for seeing that it is necessary to break up the Fluidsynth into several different Csound opcodes.

# fluidCCi

fluidCCi — Sends a MIDI controller data message to fluid.

## Syntax

```
fluidCCi iEngineNumber, iChannelNumber, iControllerNumber, iValue
```

## Description

Sends a MIDI controller data (MIDI controller number and value to use) message to a fluid engine by number on the user specified MIDI channel number.

## Initialization

*iEngineNumber* -- engine number assigned from fluidEngine

*iChannelNumber* -- MIDI channel number to which the Fluidsynth program is assigned: from 0 to 255. MIDI channels numbered 16 or higher are virtual channels.

*iControllerNumber* -- MIDI controller number to use for this message

*iValue* -- value to set for controller (usually 0-127)

## Performance

This opcode is useful for setting controller values at init time. For continuous changes, use fluidCCk.

## Examples

Here is an example of the fluidCCi opcodes. It uses the file *fluidCCi.csd* [examples/fluidCCi.csd].

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0   ;;realtime audio out and realtime midi in
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
;-o fluidCCi.wav -W   ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giengine fluidEngine
; soundfont path to manual/examples
isfnum fluidLoad "19Trumpet.sf2", giengine, 1
fluidProgramSelect giengine, 1, isfnum, 0, 56

instr 1

mididefault 60, p3
```

```
midinoteonkey p4, p5
ikey init p4
ivel init p5
fluidCCi giengine, 1, 93, 127 ;full chorus &
fluidCCi giengine, 1, 91, 127 ;full reverb!
fluidNote giengine, 1, ikey, ivel

endin

instr 99

imvol init 7
asigl, asigr fluidOut giengine
outs asigl*imvol, asigr*imvol

endin
</CsInstruments>
<CsScore>

i 1 0 5 60 100 ;play one note from score and...
i 99 0 60 ;play virtual keyboard for 60 sec.
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*fluidEngine, fluidNote, fluidLoad, fluidCCK*

More information on soundfonts in the Floss Manuals: [http://en.flossmanuals.net/csound/ch048\\_d-reading-midi-files](http://en.flossmanuals.net/csound/ch048_d-reading-midi-files)

Other information on soundfonts on Wikipedia: <http://en.wikipedia.org/wiki/Soundfont>

## Credits

Michael Gogins (gogins at pipeline dot com), Steven Yi. Thanks to Peter Hanappe for Fluidsynth.

# fluidCCk

fluidCCk — Sends a MIDI controller data message to fluid.

## Syntax

```
fluidCCk iEngineNumber, iChannelNumber, iControllerNumber, kValue
```

## Description

Sends a MIDI controller data (MIDI controller number and value to use) message to a fluid engine by number on the user specified MIDI channel number.

## Initialization

*iEngineNumber* -- engine number assigned from fluidEngine

*iChannelNumber* -- MIDI channel number to which the Fluidsynth program is assigned: from 0 to 255. MIDI channels numbered 16 or higher are virtual channels.

*iControllerNumber* -- MIDI controller number to use for this message

## Performance

*kValue* -- value to set for controller (usually 0-127)

## Examples

Here is an example of the fluidCCk opcodes. It uses the file *fluidCCk.csd* [examples/fluidCCk.csd].

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
;-o fluidCCk.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giengine fluidEngine
; soundfont path to manual/examples
isfnum fluidLoad "22Bassoon.sf2", giengine, 1
fluidProgramSelect giengine, 1, isfnum, 0, 70

instr 1

mididefault 60, p3
midinoteonkey p4, p5
ikey init p4
```

```
ivel init p5
kpan line 0, p3, 127 ;panning from left to right
fluidCCk giengine, 1, 10, kpan ;CC 10 = pan
fluidNote giengine, 1, ikey, ivel

endin

instr 99

imvol init 7
asigl, asigr fluidOut giengine
outs asigl*imvol, asigr*imvol

endin
</CsInstruments>
<CsScore>

i 1 0 4 48 100
i 1 4 2 50 120
i 1 6 1 53 80
i 1 7 1 45 70
i 1 8 1.5 48 80

i 99 0 10 ;keep instr 99 active
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*fluidEngine*, *fluidNote*, *fluidLoad*, *fluidCCi*

More information on soundfonts in the Floss Manuals: [http://en.flossmanuals.net/csound/ch048\\_d-reading-midi-files](http://en.flossmanuals.net/csound/ch048_d-reading-midi-files)

Other information on soundfonts on Wikipedia: <http://en.wikipedia.org/wiki/Soundfont>

## Credits

Michael Gogins (gogins at pipeline dot com), Steven Yi. Thanks to Peter Hanappe for Fluidsynth.

# fluidControl

fluidControl — Sends MIDI note on, note off, and other messages to a SoundFont preset.

## Syntax

```
fluidControl iengineenum, kstatus, kchannel, kdata1, kdata2
```

## Description

The fluid opcodes provide a simple Csound opcode wrapper around Peter Hanappe's Fluidsynth SoundFont2 synthesizer. This implementation accepts any MIDI note on, note off, controller, pitch bend, or program change message at k-rate. Maximum polyphony is 4096 simultaneously sounding voices. Any number of SoundFonts may be loaded and played simultaneously.

## Initialization

*iengineenum* -- engine number assigned from fluidEngine

## Performance

*kstatus* -- MIDI channel message status byte: 128 for note off, 144 for note on, 176 for control change, 192 for program change, or 224 for pitch bend.

*kchannel* -- MIDI channel number to which the Fluidsynth program is assigned: from 0 to 255. MIDI channels numbered 16 or higher are virtual channels.

*kdata1* -- For note on, MIDI key number: from 0 (lowest) to 127 (highest), where 60 is middle C. For continuous controller messages, controller number.

*kdata2* -- For note on, MIDI key velocity: from 0 (no sound) to 127 (loudest). For continuous controller messages, controller value.

Invoke fluidControl in instrument definitions that actually play notes and send control messages. Each instrument definition must consistently use one MIDI channel that was assigned to a Fluidsynth program using fluidLoad.

In this implementation, SoundFont effects such as chorus or reverb are used if and only if they are defaults for the preset. There is no means of turning such effects on or off, or of changing their parameters, from Csound.

## Examples

Here is a more complex example of the fluidsynth opcodes written by Istvan Varga. It uses the file *fluid-complex.csd* [examples/fluidcomplex.csd].

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Anna.mid is a midi file, a song by The Beatles and can be found on the internet
-odac -T -F Anna.mid;;;realtime audio I/O and midifile in
;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o fluidcomplex.wav -W ;; for file output any platform
</CsOptions>
```

```

<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
Odbfs = 1

; Example by Istvan Varga

; disable triggering of instruments by MIDI events

ichn = 1
lp1:
    massign    ichn, 0
    loop_le    ichn, 1, 16, lp1
    pgmassign  0, 0

; initialize FluidSynth

gifld    fluidEngine
gisf2    fluidLoad "sf_GMbank.sf2", gifld, 1

; k-rate version of fluidProgramSelect

opcode fluidProgramSelect_k, 0, kkkkk
    keng, kchn, ksf2, kbnk, kpre xin
    igoto    skipInit
doInit:
    fluidProgramSelect i(keng), i(kchn), i(ksf2), i(kbnk), i(kpre)
    reinit    doInit
    rireturn
skipInit:
endop

instr 1
; initialize channels
kchn    init 1
if (kchn == 1) then
lp2:
    fluidControl gifld, 192, kchn - 1, 0, 0
    fluidControl gifld, 176, kchn - 1, 7, 100
    fluidControl gifld, 176, kchn - 1, 10, 64
    loop_le    kchn, 1, 16, lp2
endif

; send any MIDI events received to FluidSynth
nxt:
kst, kch, kd1, kd2 midiin
if (kst != 0) then
    if (kst != 192) then
        fluidControl gifld, kst, kch - 1, kd1, kd2
    else
        fluidProgramSelect_k gifld, kch - 1, gisf2, 0, kd1
    endif
    kgoto nxt
endif

; get audio output from FluidSynth
ivol    init 3 ;a bit louder
aL, aR    fluidOut gifld
        outs    aL*ivol, aR*ivol
endin

</CsInstruments>
<CsScore>

i 1 0 3600

```

e

```
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*fluidEngine*, *fluidNote*, *fluidLoad*

More information on soundfonts in the Floss Manuals: [http://en.flossmanuals.net/csound/ch048\\_d-reading-midi-files](http://en.flossmanuals.net/csound/ch048_d-reading-midi-files)

Other information on soundfonts on Wikipedia: <http://en.wikipedia.org/wiki/Soundfont>

## Credits

Opcodes by Michael Gogins (gogins at pipeline dot com). Thanks to Peter Hanappe for Fluidsynth, and to Steven Yi for seeing that it is necessary to break up the Fluidsynth into several different Csound opcodes.

New in Csound5.00

# fluidEngine

fluidEngine — Instantiates a fluidsynth engine.

## Syntax

```
ienginenum fluidEngine [iReverbEnabled] [, iChorusEnabled] [,iNumChannels] [, iPolyphony]
```

## Description

Instantiates a fluidsynth engine, and returns *ienginenum* to identify the engine. *ienginenum* is passed to other other opcodes for loading and playing SoundFonts and gathering the generated sound.

## Initialization

*ienginenum* -- engine number assigned from fluidEngine.

*iReverbEnabled* -- optionally set to 0 to disable any reverb effect in the loaded SoundFonts.

*iChorusEnabled* -- optionally set to 0 to disable any chorus effect in the loaded SoundFonts.

*iNumChannels* -- number of channels to use; range is 16-256 and Csound default is 256 (Fluidsynth's default is 16).

*iPolyphony* -- number of voices to be played in parallel; range is 16-4096 and Csound default is 4096 (Fluidsynth's default is 256). Note: this is not the number of notes played at the same time as a single note may use create multiple voices depending on instrument zones and velocity/key of played note.

## Examples

Here is example of the fluidsynth opcodes using 2 engines. It uses the file *fluidEngine.csd* [examples/fluidEngine.csd] and *midichn\_advanced.mid* [examples/midichn\_advanced.mid].

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -F midichn_advanced.mid ;;realtime audio out and midifile in
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o fluidEngine.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

; LOAD SOUNDFONTS
gienginenum1 fluidEngine
gienginenum2 fluidEngine
isfnum1 fluidLoad "sf_GMbank.sf2", gienginenum1, 1
; Piano 2, program 1, channel 1
fluidProgramSelect gienginenum1, 1, isfnum1, 0, 1
; Piano 3, program 2, channel 2
fluidProgramSelect gienginenum1, 2, isfnum1, 0, 2
isfnum2 fluidLoad "19Trumpet.sf2", gienginenum2, 1
```

```

        ; Trumpet, program 56, channel 3
    fluidProgramSelect gienginenum2, 3, isfnum2, 0, 56

;Look for midifile in folder manual/examples
;midichn_advanced.mid" sends notes to the soundfonts

instr 1 ; GM soundfont
    ; INITIALIZATION
    mididefault 60, p3 ; Default duration of 60 -- overridden by score.
    midinoteonkey p4, p5 ; Channels MIDI input to pfields.
    ; Use channel assigned in fluidload.
    ichannel = 1
    ikey = p4
    ivelocity = p5
    fluidNote gienginenum1, ichannel, ikey, ivelocity
endin

instr 2 ; GM soundfont
    ; INITIALIZATION
    mididefault 60, p3 ; Default duration of 60 -- overridden by score.
    midinoteonkey p4, p5 ; Channels MIDI input to pfields.
    ; Use channel assigned in fluidload.
    ichannel = 2
    ikey = p4
    ivelocity = p5
    fluidNote gienginenum1, ichannel, ikey, ivelocity
endin

instr 3 ; Trumpet
    ; INITIALIZATION
    mididefault 60, p3 ; Default duration of 60 -- overridden by score.
    midinoteonkey p4, p5 ; Channels MIDI input to pfields.
    ; Use channel assigned in fluidload.
    ichannel = 3
    ikey = p4
    ivelocity = p5
    fluidNote gienginenum2, ichannel, ikey, ivelocity
endin

; COLLECT AUDIO FROM ALL SOUND FONTS

instr 100 ; Fluidsynth output

    iamplitude1 = 7
    iamplitude2 = 7

; AUDIO
    aleft1, aright1 fluidOut gienginenum1
    aleft2, aright2 fluidOut gienginenum2
    outs (aleft1 * iamplitude1) + (aleft2 * iamplitude2), \
        (aright1 * iamplitude1) + (aright2 * iamplitude2)

endin
</CsInstruments>
<CsScore>

i 1 0 3 60 100
i 2 1 3 60 100
i 3 3 3 63 100
i 100 0 10 ;run for 10 seconds
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*fluidNote, fluidLoad*

More information on soundfonts in the Floss Manuals: [http://en.flossmanuals.net/csound/ch048\\_d-reading-midi-files](http://en.flossmanuals.net/csound/ch048_d-reading-midi-files)

Other information on soundfonts on Wikipedia: <http://en.wikipedia.org/wiki/Soundfont>

## Credits

Michael Gogins (gogins at pipeline dot com), Steven Yi. Thanks to Peter Hanappe for Fluidsynth.

Optional *iNumChannels* and *iPolyphony* parameters added in 5.07

# fluidLoad

`fluidLoad` — Loads a SoundFont into a `fluidEngine`, optionally listing SoundFont contents.

## Syntax

```
isfnum fluidLoad soundfont, ienginenum[, ilistpresets]
```

## Description

Loads a SoundFont into an instance of a `fluidEngine`, optionally listing banks and presets for SoundFont.

## Initialization

*isfnum* -- number assigned to just-loaded soundfont.

*soundfont* -- string specifying a SoundFont filename. Note that any number of SoundFonts may be loaded (obviously, by different invocations of `fluidLoad`).

*ienginenum* -- engine number assigned from `fluidEngine`

*ilistpresets* -- optional, if specified, lists all Fluidsynth programs for the just-loaded SoundFont. A Fluidsynth program is a combination of SoundFont ID, bank number, and preset number that is assigned to a MIDI channel.

## Performance

Invoke `fluidLoad` in the orchestra header, any number of times. The same SoundFont may be invoked to assign programs to MIDI channels any number of times; the SoundFont is only loaded the first time.

## Examples

Here is an example of the `fluidLoad` opcode. It uses the file `fluidLoad.csd` [examples/fluidLoad.csd] and `07AcousticGuitar.sf2` [examples/07AcousticGuitar.sf2].

### Example 312. Example of the `fluidLoad` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac  --rtmidi=virtual  -M0  ;;realtime audio out and realtime midi in
;-iadc  ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
;-o fluidLoad.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
```

```
Odbfs = 1

giengine fluidEngine
; soundfont path to manual/examples
isfnum fluidLoad "07AcousticGuitar.sf2", giengine, 1
fluidProgramSelect giengine, 1, isfnum, 0, 0

instr 1

mididefault 60, p3
midinoteonkey p4, p5
ikey init p4
ivel init p5
fluidNote giengine, 1, ikey, ivel

endin

instr 99

imvol init 7
asigl, asigr fluidOut giengine
outs asigl*imvol, asigr*imvol

endin
</CsInstruments>
<CsScore>

i 1 0 2 60 100 ;play one note from score and...
i 99 0 60 ;play virtual keyboard for 60 sec.
e

</CsScore>
</CsoundSynthesizer>
```

The output should include a line like this:

```
SoundFont: 1 Bank: 0 Preset: 0 Seagul Acoustic Git
```

## See Also

*fluidEngine, fluidNote*

More information on soundfonts in the Floss Manuals: [http://en.flossmanuals.net/csound/ch048\\_d-reading-midi-files](http://en.flossmanuals.net/csound/ch048_d-reading-midi-files)

Other information on soundfonts on Wikipedia: <http://en.wikipedia.org/wiki/Soundfont>

## Credits

Michael Gogins (gogins at pipeline dot com), Steven Yi. Thanks to Peter Hanappe for Fluidsynth.

New in Csound5.00

# fluidNote

fluidNote — Plays a note on a channel in a fluidSynth engine.

## Syntax

```
fluidNote ienginenum, ichannelnum, imidikey, imidivel
```

## Description

Plays a note at *imidikey* pitch and *imidivel* velocity on *ichannelnum* channel of number *ienginenum* fluidEngine.

## Initialization

*ienginenum* -- engine number assigned from fluidEngine

*ichannelnum* -- which channel number to play a note on in the given fluidEngine

*imidikey* -- MIDI key for note (0-127)

*imidivel* -- MIDI velocity for note (0-127)

## Examples

Here is an example of the fluidNote opcode. It uses the file *fluidNote.csd* [examples/fluidNote.csd] and *19Trumpet.sf2* [examples/19Trumpet.sf2].

### Example 313. Example of the fluidNote opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac  -+rtmidi=virtual  -M0  ;;realtime audio out and realtime midi in
;-iadc  ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
;-o fluidNote.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giengine fluidEngine
; soundfont path to manual/examples
isfnum fluidLoad "19Trumpet.sf2", giengine, 1
fluidProgramSelect giengine, 1, isfnum, 0, 56

instr 1
```

```
mididefault 60, p3
midinoteonkey p4, p5
ikey init p4
ivel init p5
fluidNote giengine, 1, ikey, ivel

endin

instr 99

imvol init 7
asigl, asigr fluidOut giengine
outs asigl*imvol, asigr*imvol

endin
</CsInstruments>
<CsScore>

i 1 0 2 60 100 ;play one note from score and...
i 99 0 60 ;play virtual keyboard for 60 sec.
e

</CsScore>
</CsoundSynthesizer>
```

The output should include a line like this:

```
SoundFont: 1 Bank: 0 Preset: 56 Trumpet metallic
```

## See Also

*fluidEngine, fluidLoad*

More information on soundfonts in the Floss Manuals: [http://en.flossmanuals.net/csound/ch048\\_d-reading-midi-files](http://en.flossmanuals.net/csound/ch048_d-reading-midi-files)

Other information on soundfonts on Wikipedia: <http://en.wikipedia.org/wiki/Soundfont>

## Credits

Michael Gogins (gogins at pipeline dot com), Steven Yi. Thanks to Peter Hanappe for Fluidsynth.

# fluidOut

fluidOut — Outputs sound from a given fluidEngine

## Syntax

```
aleft, aright fluidOut ienginenum
```

## Description

Outputs the sound from a fluidEngine.

## Initialization

*ienginenum* -- engine number assigned from fluidEngine

## Performance

*aleft* -- Left channel audio output.

*aright* -- Right channel audio output.

Invoke fluidOut in an instrument definition numbered higher than any fluidcontrol instrument definitions. All SoundFonts used in the fluidEngine numbered *ienginenum* send their audio output to this one opcode. Send a note with an indefinite duration to this instrument to turn the SoundFonts on for as long as required.

## Examples

Here is an example of the fluidOut opcode with two fluidOuts. It uses the file *fluidOut.csd*, *01hpschd.sf2* [examples/01hpschd.sf2] and *[examples/fluidOut.csd]22Bassoon.sf2* [examples/22Bassoon.sf2].

### Example 314. Example of the fluidOut opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac  --rtmidi=virtual -M0  ;;realtime audio out and realtime midi in
;-iadc  ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
;-o fluidOut.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
```

```

giengine1 fluidEngine
; soundfont path to manual/examples
isfnum1 fluidLoad "01hpschd.sf2", giengine1, 1
fluidProgramSelect giengine1, 1, isfnum1, 0, 0

giengine2 fluidEngine
; soundfont path to manual/examples
isfnum2 fluidLoad "22Bassoon.sf2", giengine2, 1
fluidProgramSelect giengine2, 1, isfnum2, 0, 70

instr 1

mididefault 60, p3
midinoteonkey p4, p5
ikey init p4
ivel init p5
fluidNote giengine1, 1, ikey, ivel

endin

instr 2

mididefault 60, p3
midinoteonkey p4, p5
ikey init p4
ivel init p5
fluidNote giengine2, 1, ikey, ivel

endin

instr 98

imvol init 7
asigl, asigr fluidOut giengine1
outs asigl*imvol, asigr*imvol

endin

instr 99

imvol init 4
asigl, asigr fluidOut giengine2 ;add a stereo flanger
adelL linseg 0, p3*.5, 0.02, p3*.5, 0 ;max delay time =20ms
adelR linseg 0.02, p3*.5, 0, p3*.5, 0.02 ;max delay time =20ms
asigL flanger asigl, adelL, .6
asigR flanger asigr, adelR, .6
outs asigL*imvol, asigR*imvol

endin
</CsInstruments>
<CsScore>

i 1 0 2 60 100 ;play one note of instr 1
i 2 2 2 60 100 ;play another note of instr 2 and...
i 98 0 60 ;play virtual keyboard for 60 sec.
i 99 0 60
e
</CsScore>
</CsoundSynthesizer>

```

The output should include lines like these:

```

chnl 1 using instr 1
chnl 2 using instr 2

SoundFont: 1 Bank: 0 Preset: 0 Harpsichord I-8
SoundFont: 1 Bank: 0 Preset: 70 Ethan Bassoon mono

```

## See Also

*fluidEngine, fluidNote, fluidLoad*

More information on soundfonts in the Floss Manuals: [http://en.flossmanuals.net/csound/ch048\\_d-reading-midi-files](http://en.flossmanuals.net/csound/ch048_d-reading-midi-files)

Other information on soundfonts on Wikipedia: <http://en.wikipedia.org/wiki/Soundfont>

## Credits

Michael Gogins (gogins at pipeline dot com), Steven Yi. Thanks to Peter Hanappe for Fluidsynth.

New in Csound5.00

# fluidProgramSelect

fluidProgramSelect — Assigns a preset from a SoundFont to a channel on a fluidEngine.

## Syntax

```
fluidProgramSelect iengineum, ichannelnum, isfnum, ibanknum, ipresetnum
```

## Description

Assigns a preset from a SoundFont to a channel on a fluidEngine.

## Initialization

*iengineum* -- engine number assigned from fluidEngine

*ichannelnum* -- which channel number to use for the preset in the given fluidEngine

*isfnum* -- number of the SoundFont from which the preset is assigned

*ibanknum* -- number of the bank in the SoundFont from which the preset is assigned

*ipresetnum* -- number of the preset to assign

## Examples

Here is an example of the fluidProgramSelect opcode. It uses the file *fluidProgramSelect.csd* [examples/fluidProgramSelect.csd].

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac  -+rtmidi=virtual  -M0  ;;realtime audio out and realtime midi in
-iadc  ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
;-o fluidProgramSelect.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giengine fluidEngine
; soundfont path to manual/examples
isfnum fluidLoad "19Trumpet.sf2", giengine, 1
fluidProgramSelect giengine, 1, isfnum, 0, 56

instr 1

mididefault 60, p3
midinoteonkey p4, p5
ikey init p4
ivel init p5
fluidNote giengine, 1, ikey, ivel
```

```

endin

instr 99

imvol  init 7
asigl, asigr fluidOut giengine
      outs asigl*imvol, asigr*imvol

endin
</CsInstruments>
<CsScore>

i 1 0 2 60 100 ;play one note from score and...
i 99 0 60      ;play virtual keyboard for 60 sec.
e

</CsScore>
</CsoundSynthesizer>

```

Here is another more complex example of the fluidsynth opcodes written by Istvan Varga. It uses the file *fluidcomplex.csd* [examples/fluidcomplex.csd].

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
;Anna.mid is a midi file, a song by The Beatles and can be found on the internet
-odac -T -F Anna.mid;;;realtime audio I/O and midifile in
;-iadc      ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o fluidcomplex.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

; Example by Istvan Varga

; disable triggering of instruments by MIDI events

ichn = 1
lp1:
    massign  ichn, 0
    loop_le  ichn, 1, 16, lp1
    pgmassign 0, 0

; initialize FluidSynth

gifld  fluidEngine
gisf2  fluidLoad "sf_GMbank.sf2", gifld, 1

; k-rate version of fluidProgramSelect

opcode fluidProgramSelect_k, 0, kkkkk
    keng, kchn, ksf2, kbnk, kpre xin
    igoto      skipInit
doInit:
    fluidProgramSelect i(keng), i(kchn), i(ksf2), i(kbnk), i(kpre)
    reinit  doInit
    rireturn
skipInit:
endop

```

```
instr 1
; initialize channels
kchn init 1
if (kchn == 1) then
lp2:
    fluidControl gifld, 192, kchn - 1, 0, 0
    fluidControl gifld, 176, kchn - 1, 7, 100
    fluidControl gifld, 176, kchn - 1, 10, 64
    loop_le kchn, 1, 16, lp2
endif

; send any MIDI events received to FluidSynth
nxt:
kst, kch, kd1, kd2 midiin
if (kst != 0) then
    if (kst != 192) then
        fluidControl gifld, kst, kch - 1, kd1, kd2
    else
        fluidProgramSelect_k gifld, kch - 1, gisf2, 0, kd1
    endif
    kgoto nxt
endif

; get audio output from FluidSynth
ivol init 3 ;a bit louder
aL, aR fluidOut gifld
outs aL*ivol, aR*ivol
endin

</CsInstruments>
<CsScore>

i 1 0 3600
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*fluidEngine, fluidNote, fluidLoad*

More information on soundfonts in the Floss Manuals: [http://en.flossmanuals.net/csound/ch048\\_d-reading-midi-files](http://en.flossmanuals.net/csound/ch048_d-reading-midi-files)

Other information on soundfonts on Wikipedia: <http://en.wikipedia.org/wiki/Soundfont>

## Credits

Michael Gogins (gogins at pipeline dot com), Steven Yi. Thanks to Peter Hanappe for Fluidsynth.

# fluidSetInterpMethod

fluidSetInterpMethod — Set interpolation method for channel in Fluid Engine

## Syntax

```
fluidSetInterpMethod ienginenum, ichannelnum, iInterpMethod
```

## Description

Set interpolation method for channel in Fluid Engine. Lower order interpolation methods will render faster at lower fidelity while higher order interpolation methods will render slower at higher fidelity. Default interpolation for a channel is 4th order interpolation.

## Initialization

*ienginenum* -- engine number assigned from fluidEngine

*ichannelnum* -- which channel number to use for the preset in the given fluidEngine

*iInterpMethod* -- interpolation method, can be any of the following

- 0 -- No Interpolation
- 1 -- Linear Interpolation
- 4 -- 4th Order Interpolation (Default)
- 7 -- 7th Order Interpolation (Highest)

## Examples

Here is an example of the fluidSetInterpMethod opcode. It uses the file *fluidSetInterpMethod.csd* [examples/fluidSetInterpMethod.csd] and *07AcousticGuitar.sf2* [examples/07AcousticGuitar.sf2].

### Example 315. Example of the fluidSetInterpMethod opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out and realtime midi in
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
;-o fluidSetInterpMethod.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
```

```
giengine fluidEngine
; soundfont path to manual/examples
isfnum fluidLoad "07AcousticGuitar.sf2", giengine, 1
fluidProgramSelect giengine, 1, isfnum, 0, 0

instr 1

mididefault 60, p3
midinoteonkey p4, p5
ikey init p4
ivel init p5
iInterpMethod = p6
fluidSetInterpMethod giengine, 1, iInterpMethod
fluidNote giengine, 1, ikey, ivel

endin

instr 99

imvol init 7
asigl, asigr fluidOut giengine
outs asigl*imvol, asigr*imvol

endin
</CsInstruments>
<CsScore>
;hear the difference
i 1 0 2 60 120 0 ;no interpolation
i 1 3 2 72 120 0
i 1 6 2 60 120 7 ;7th order interpolation
i 1 9 2 72 120 7

i 99 0 12

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*fluidEngine*

More information on soundfonts in the Floss Manuals: [http://en.flossmanuals.net/csound/ch048\\_d-reading-midi-files](http://en.flossmanuals.net/csound/ch048_d-reading-midi-files)

Other information on soundfonts on Wikipedia: <http://en.wikipedia.org/wiki/Soundfont>

## Credits

Author: Steven Yi

New in version 5.07

# FLvalue

FLvalue — Shows the current value of a FLTK valuator.

## Description

*FLvalue* shows current the value of a valuator in a text field.

## Syntax

```
ihandle FLvalue "label", iwidth, iheight, ix, iy
```

## Initialization

*ihandle* -- handle value (an integer number) that unequivocally references the corresponding valuator. It can be used for the *idisp* argument of a valuator.

*"label"* -- a double-quoted string containing some user-provided text, placed near the corresponding widget.

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

## Performance

*FLvalue* shows the current values of a valuator in a text field. It outputs *ihandle* that can then be used for the *idisp* argument of a valuator (see the *FLTK Valuators section*). In this way, the values of that valuator will be dynamically be shown in a text field.



### Note

Note that *FLvalue* is not a valuator and its value cannot be modified. The value for an *FLvalue* widget should be set only by other widgets, and NOT from *FLsetVal* or *FLsetVal\_i* since this can cause Csound to crash.

## Examples

Here is an example of the FLvalue opcode. It uses the file *FLvalue.csd* [examples/FLvalue.csd].

### Example 316. Example of the FLvalue opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
```

```
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in   No messages
-odac         -iadc      -d          ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o FLvalue.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Using the opcode flvalue to display the output of a slider
sr = 44100
kr = 441
ksmps = 100
nchnls = 1

FLpanel "Value Display Box", 900, 200, 50, 50
; Width of the value display box in pixels
iwidth = 50
; Height of the value display box in pixels
iheight = 20
; Distance of the left edge of the value display
; box from the left edge of the panel
ix = 65
; Distance of the top edge of the value display
; box from the top edge of the panel
iy = 55

idisp FLvalue "Hertz", iwidth, iheight, ix, iy
gkfreq, ihandle FLslider "Frequency", 200, 5000, -1, 5, idisp, 750, 30, 125, 50
FLsetVal_i 500, ihandle
; End of panel contents
FLpanelEnd
; Run the widget thread!
FLrun

instr 1
iamp = 15000
ifn = 1
asig oscili iamp, gkfreq, ifn
out asig
endin

</CsInstruments>
<CsScore>

; Function table that defines a single cycle
; of a sine wave.
f 1 0 1024 10 1

; Instrument 1 will play a note for 1 hour.
i 1 0 3600
e

</CsScore>
</CsSoundSynthesizer>
```

## See Also

*FLbox, FLbutBank, FLbutton, FLprintk, FLprintk2*

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.

# FLvkeybd

FLvkeybd — An FLTK widget opcode that creates a virtual keyboard widget.

## Description

An FLTK widget opcode that creates a virtual keyboard widget. This must be used in conjunction with the virtual midi keyboard driver for this to operate correctly. The purpose of this opcode is for making demo versions of MIDI orchestras with the virtual keyboard embedded within the main window.



### Note

The widget version of the virtual keyboard does not include the MIDI sliders found in the full window version of the virtual keyboard.

## Syntax

```
FLvkeybd "keyboard.map", iwidth, iheight, ix, iy
```

## Initialization

*“keyboard.map”* -- a double-quoted string containing the keyboard map to use. An empty string (“”) may be used to use the default bank/channel name values. See [Virtual Midi Keyboard](#) for more information on keyboard mappings.

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* -- horizontal position of upper left corner of the keyboard, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the keyboard, relative to the upper left corner of corresponding window (expressed in pixels).



### Note

The standard width and height for the virtual keyboard is 624x120 for the dialog version that is shown when FLvkeybd is not used.

## See Also

*FLbutton*, *FLbox*, *FLbutBank*, *FLprintk*, *FLprintk2*, *FLvalue*

## Credits

Author: Steven Yi

New in version 5.05

# FLvslidBnk

FLvslidBnk — A FLTK widget containing a bank of vertical sliders.

## Description

*FLvslidBnk* is a widget containing a bank of vertical sliders.

## Syntax

```
FLvslidBnk "names", inumsliders [, ioutable] [, iwidth] [, iheight] [, ix] \  
[, iy] [, itypetable] [, iexptable] [, istart_index] [, iminmaxtable]
```

## Initialization

*“names”* -- a double-quoted string containing the names of each slider. Each slider can have a different name. Separate each name with “@” character, for example: “frequency@amplitude@cutoff”. It is possible to not provide any name by giving a single space “ ”. In this case, the opcode will automatically assign a progressive number as a label for each slider.

*inumsliders* -- the number of sliders.

*ioutable* (optional, default=0) -- number of a previously-allocated table in which to store output values of each slider. The user must be sure that table size is large enough to contain all output cells, otherwise a segfault will crash Csound. By assigning zero to this argument, the output will be directed to the zak space in the k-rate zone. In this case, the zak space must be previously allocated with the *zakinit* opcode and the user must be sure that the allocation size is big enough to cover all sliders. The default value is zero (i.e. store output in zak space).

*istart\_index* (optional, default=0) -- an integer number referring to a starting offset of output cell locations. It can be positive to allow multiple banks of sliders to output in the same table or in the zak space. The default value is zero (no offset).

*iminmaxtable* (optional, default=0) -- number of a previously-defined table containing a list of min-max pairs, referred to each slider. A zero value defaults to the 0 to 1 range for all sliders without necessity to provide a table. The default value is zero.

*iexptable* (optional, default=0) -- number of a previously-defined table containing a list of identifiers (i.e. integer numbers) provided to modify the behaviour of each slider independently. Identifiers can assume the following values:

- -1 -- exponential curve response
- 0 -- linear response
- number > than 0 -- follow the curve of a previously-defined table to shape the response of the corresponding slider. In this case, the number corresponds to table number.

You can assume that all sliders of the bank have the same response curve (exponential or linear). In this case, you can assign -1 or 0 to *iexptable* without worrying about previously defining any table. The default value is zero (all sliders have a linear response, without having to provide a table).

*ityetable* (optional, default=0) -- number of a previously-defined table containing a list of identifiers (i.e. integer numbers) provided to modify the aspect of each individual slider independently. Identifiers can assume the following values:

- 0 = Nice slider
- 1 = Fill slider
- 3 = Normal slider
- 5 = Nice slider
- 7 = Nice slider with down-box

You can assume that all sliders of the bank have the same aspect. In this case, you can assign a negative number to *ityetable* without worrying about previously defining any table. Negative numbers have the same meaning of the corresponding positive identifiers with the difference that the same aspect is assigned to all sliders. You can also assign a random aspect to each slider by setting *ityetable* to a negative number lower than -7. The default value is zero (all sliders have the aspect of nice sliders, without having to provide a table).

You can add 20 to a value inside the table to make the slider "plastic", or subtract 20 if you want to set the value for all widgets without defining a table (e.g. -21 to set all sliders types to Plastic Fill slider).

*iwidth* (optional) -- width of the rectangular area containing all sliders of the bank, excluding text labels, that are placed below that area.

*iheight* (optional) -- height of the rectangular area containing all sliders of the bank, excluding text labels, that are placed below that area.

*ix* (optional) -- horizontal position of the upper left corner of the rectangular area containing all sliders belonging to the bank. You have to leave enough space, below that rectangle, in order to make sure labels of sliders to be visible. This is because the labels themselves are external to the rectangular area.

*iy* (optional) -- vertical position of the upper left corner of the rectangular area containing all sliders belonging to the bank. You have to leave enough space, below that rectangle, in order to make sure labels of sliders to be visible. This is because the labels themselves are external to the rectangular area.

## Performance

There are no k-rate arguments, even if cells of the output table (or the zak space) are updated at k-rate.

*FLvslidBnk* is a widget containing a bank of vertical sliders. Any number of sliders can be placed into the bank (*inumsliders* argument). The output of all sliders is stored into a previously allocated table or into the zak space (*ioutable* argument). It is possible to determine the first location of the table (or of the zak space) in which to store the output of the first slider by means of *istart\_index* argument.

Each slider can have an individual label that is placed below it. Labels are defined by the "names" argument. The output range of each slider can be individually set by means of an external table (*iminmaxtable* argument). The curve response of each slider can be set individually, by means of a list of identifiers placed in a table (*ixptable* argument). It is possible to define the aspect of each slider independently or to make all sliders have the same aspect (*ityetable* argument).

The *iwidth*, *iheight*, *ix*, and *iy* arguments determine width, height, horizontal and vertical position of the rectangular area containing sliders. Notice that the label of each slider is placed below them and is not

included in the rectangular area containing sliders. So the user should leave enough space below the bank by assigning a proper *iy* value in order to leave labels visible.

*FLvslidBnk* is identical to *FLslidBnk* except it contains vertical sliders instead of horizontal. Since the width of each single slider is often small, it is recommended to leave only a single space in the names string (“ ”), in this case each slider will be automatically numbered.



## IMPORTANT!

Notice that the tables used by *FLvslidBnk* must be created with the *ftgen* opcode and placed in the orchestra before the corresponding valuator. They can not be placed in the score. This is because tables placed in the score are created later than the initialization of the opcodes placed in the header section of the orchestra.

## Examples

Here is an example of the *FLvslidBnk* opcode. It uses the file *FLvslidBnk.csd* [examples/FLvslidBnk.csd].

### Example 317. Example of the *FLvslidBnk* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      ;;RT audio I/O
</CsOptions>
<CsInstruments>

sr = 44100
kr = 441
ksmps = 100
nchnls = 1

gitypetable ftgen 0, 0, 8, -2, 1, 1, 3, 3, 5, 5, 7, 7
giouttable ftgen 0, 0, 8, -2, 0, 0.2, 0.3, 0.4, 0.5, 0.6, 0.8, 1

FLpanel "Slider Bank", 400, 400, 50, 50
;Number of sliders
inum = 8
; Table to store output
iouttable = giouttable
; Width of the slider bank in pixels
iwidth = 350
; Height of the slider in pixels
iheight = 160
; Distance of the left edge of the slider
; from the left edge of the panel
ix = 30
; Distance of the top edge of the slider
; from the top edge of the panel
iy = 10
; Table containing fader types
itypetable = gitypetable
FLvslidBnk "1@2@3@4@5@6@7@8@9@10@11@12@13@14@15@16", 16 , iouttable , iwidth , iheight , ix \
, iy , itypetable
FLvslidBnk " " , inum , iouttable , iwidth , iheight , ix \
, iy + 200 , -23
; End of panel contents
```

```
FLpanelEnd
; Run the widget thread!
FLrun

instr 1
;Dummy instrument
endin

</CsInstruments>
<CsScore>

; Instrument 1 will play a note for 1 hour.
i 1 0 3600
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*FLslider, FLslidBnk*

## Credits

Author: Gabriel Maldonado

New in version 5.06

# FLvslidBnk2

FLvslidBnk2 — A FLTK widget containing a bank of vertical sliders.

## Description

*FLvslidBnk2* is a widget containing a bank of vertical sliders.

## Syntax

```
FLvslidBnk2 "names", inumsliders, ioutable, iconfigtable [,iwidth, iheight, ix, iy, istart_index]
```

## Initialization

*"names"* -- a double-quoted string containing the names of each slider. Each slider can have a different name. Separate each name with "@" character, for example: "frequency@amplitude@cutoff". It is possible to not provide any name by giving a single space " ". In this case, the opcode will automatically assign a progressive number as a label for each slider.

*inumsliders* -- the number of sliders.

*ioutable* (optional, default=0) -- number of a previously-allocated table in which to store output values of each slider. The user must be sure that table size is large enough to contain all output cells, otherwise a segfault will crash Csound. By assigning zero to this argument, the output will be directed to the zak space in the k-rate zone. In this case, the zak space must be previously allocated with the *zakinit* opcode and the user must be sure that the allocation size is big enough to cover all sliders. The default value is zero (i.e. store output in zak space).

*iconfigtable* -- in the *FLslidBnk2* and *FLvslidBnk2* opcodes, this table replaces *iminmaxtable*, *iehtable* and *istyletable*, all these parameters being placed into a single table. This table has to be filled with a group of 4 parameters for each slider in this way:

min1, max1, exp1, style1, min2, max2, exp2, style2, min3, max3, exp3, style3 etc.

for example using GEN02 you can type:

```
inum ftgen 1,0,256, -2, 0,1,0,1, 100, 5000, -1, 3, 50, 200, -1, 5,..... [etcetera]
```

In this example the first slider will be affected by the [0,1,0,1] parameters (the range will be 0 to 1, it will have linear response, and its aspect will be a fill slider), the second slider will be affected by the [100,5000,-1,3] parameters (the range is 100 to 5000, the response is exponential and the aspect is a normal slider), the third slider will be affected by the [50,200,-1,5] parameters (the range is 50 to 200, the behavior exponential, and the aspect is a nice slider), and so on.

*iwidth* (optional) -- width of the rectangular area containing all sliders of the bank, excluding text labels, that are placed below that area.

*iheight* (optional) -- height of the rectangular area containing all sliders of the bank, excluding text labels, that are placed below that area.

*ix* (optional) -- horizontal position of the upper left corner of the rectangular area containing all sliders belonging to the bank. You have to leave enough space, below that rectangle, in order to make sure labels of sliders to be visible. This is because the labels themselves are external to the rectangular area.

*iy* (optional) -- vertical position of the upper left corner of the rectangular area containing all sliders belonging to the bank. You have to leave enough space, below that rectangle, in order to make sure labels of sliders to be visible. This is because the labels themselves are external to the rectangular area.

*istart\_index* (optional, default=0) -- an integer number referring to a starting offset of output cell locations. It can be positive to allow multiple banks of sliders to output in the same table or in the zak space. The default value is zero (no offset).

## Performance

There are no k-rate arguments, even if cells of the output table (or the zak space) are updated at k-rate.

*FLvslidBnk2* is a widget containing a bank of vertical sliders. Any number of sliders can be placed into the bank (*inumsliders* argument). The output of all sliders is stored into a previously allocated table or into the zak space (*ioutable* argument). It is possible to determine the first location of the table (or of the zak space) in which to store the output of the first slider by means of *istart\_index* argument.

Each slider can have an individual label that is placed below it. Labels are defined by the “*names*” argument. The output range of each slider can be individually set by means of the *min* and *max* values inside the *iconfigtable* table. The curve response of each slider can be set individually, by means of a list of identifiers placed in the *iconfigtable* table (*exp* argument). It is possible to define the aspect of each slider independently or to make all sliders have the same aspect (*style* argument in the *iconfigtable* table).

The *iwidth*, *iheight*, *ix*, and *iy* arguments determine width, height, horizontal and vertical position of the rectangular area containing sliders. Notice that the label of each slider is placed below them and is not included in the rectangular area containing sliders. So the user should leave enough space below the bank by assigning a proper *iy* value in order to leave labels visible.

*FLvslidBnk2* is identical to *FLslidBnk2* except it contains vertical sliders instead of horizontal. Since the width of each single slider is often small, it is recommended to leave only a single space in the names string (“ ”), in this case each slider will be automatically numbered.



### IMPORTANT!

Notice that the tables used by *FLvslidBnk2* must be created with the *ftgen* opcode and placed in the orchestra before the corresponding valuator. They can not placed in the score. This is because tables placed in the score are created later than the initialization of the opcodes placed in the header section of the orchestra.

## See Also

*FLslider*, *FLslidBnk*, *FLslidBnk2*, *FLvslidBnk2*

## Credits

Author: Gabriel Maldonado

New in version 5.06

# FLxyin

FLxyin — Senses the mouse cursor position in a user-defined area inside an FLpanel.

## Description

Similar to *xyin*, sense the mouse cursor position in a user-defined area inside an FLpanel.

## Syntax

```
koutx, kouty, kinside FLxyin ioutx_min, ioutx_max, iouty_min, iouty_max, \  
iwindx_min, iwindx_max, iwindy_min, iwindy_max [, iexpx, iexpy, ioutx, iouty]
```

## Initialization

*ioutx\_min, ioutx\_max* - the minimum and maximum limits of the interval to be output (X or horizontal axis).

*iouty\_min, iouty\_max* - the minimum and maximum limits of the interval to be output (Y or vertical axis).

*iwindx\_min, iwindx\_max* - the X coordinate of the horizontal edges of the sensible area, relative to the FLpanel, in pixels.

*iwindy\_min, iwindy\_max* - the Y coordinates of the vertical edges of the sensible area, relative to the FLpanel, in pixels.

*iexpx, iexpy* - (optional) integer numbers denoting the behavior of the x or y output: 0 -> output is linear; -1 -> output is exponential; any other number indicates the number of an existing table that is used for indexing. With a positive value for table number, linear interpolation is provided in table indexing. A negative table number suppresses interpolation. Notice that in normal operations, the table should be normalized and unipolar (i.e. all table elements should be in the zero to one range). In this case all table elements will be rescaled according to *imax* and *imin*. Anyway, it is possible to use non-normalized tables (created with a negative table number, that can contain elements of any value), in order to access the actual values of table elements, without rescaling, by assigning 0 to *iout\_min* and 1 to *iout\_max*.

*ioutx, iouty* - (optional) initial output values.

## Performance

*koutx, kouty* - output values, scaled according to user choices.

*kinside* - a flag that informs if the mouse cursor falls out of the rectangle of the user-defined area. If it is out of the area, *kinside* is set to zero.

*FLxyin* senses the mouse cursor position in a user-defined area inside an *FLpanel*. When *FLxyin* is called, the position of the mouse within the chosen area is returned at *k*-rate. It is possible to define the sensible area, as well the minimum and maximum values corresponding to the minimum and maximum mouse positions. Mouse buttons don't need to be pressed to make *FLxyin* to operate. It is able to function correctly even if other widgets (present in the *FLpanel*) overlap the sensible area.

*FLxyin* unlike most other FLTK opcodes can't be used inside the header, since it is not a widget. It is just a definition of an area for mouse sensing within an FLTK panel.

## Examples

Here is an example of the FLxyin opcode. It uses the file *FLxyin.csd* [examples/FLxyin.csd].

### Example 318. Example of the FLxyin opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in
-odac         -iadc   ;;RT audio I/O
</CsOptions>
<CsInstruments>

sr=48000
ksmps=128
nchnls=2

; Example by Andres Cabrera 2007

FLpanel "FLxyin", 200, 100, -1, -1, 3
FLpanelEnd
FLrun

instr 1
  koutx, kouty, kinside FLxyin 0, 10, 100, 1000, 10, 190, 10, 90
  aout buzz 10000, kouty, koutx, 1
  printk2 koutx
  outs aout, aout
endin

</CsInstruments>

<CsScore>
f 1 0 1024 10 1
i 1 0 3600

e

</CsScore>
</CsoundSynthesizer>
```

Here is another example of the FLxyin opcode. It uses the file *FLxyin-2.csd* [examples/FLxyin-2.csd].

### Example 319. Example of the FLxyin opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in   No messages
-odac         -iadc     -d   ;;RT audio I/O
</CsOptions>
<CsInstruments>

sr=44100
kr=441
ksmps=100
```

```
nchnls=2

; Example by Gabriel Maldonado

FLpanel "Move the mouse inside this panel to hear the effect",400,400
FLpanel_end
FLrun

instr 1

k1, k2, kinside FLxyin 50, 1000, 50, 1000, 100, 300, 50, 250, -2,-3
;if k1 <= 50 || k1 >=5000 || k2 <=100 || k2 >= 8000 kgoto end ; if cursor is outside bounds, then don't

a1 oscili 3000, k1, 1
a2 oscili 3000, k2, 1

outs a1,a2
printk2 k1
printk2 k2, 10
printk2 kinside, 20
end:
endin

</CsInstruments>
<CsScore>

f1 0 1024 10 1
f2 0 17 19 1 1 90 1
f3 0 17 19 2 1 90 1
i1 0 3600

</CsScore>
</CsoundSynthesizer>
```

## See Also

*FLpanel*

## Credits

Author: Gabriel Maldonado

New in version 5.06

# fmanal

fmanal — AM/FM analysis from quadrature signal.

## Description

This opcode attempts to extract the AM and FM signals off a quadrature signal (e.g. from a Hilbert transform).

## Syntax

am, af **fmanal** are, aim

## Performance

*are* -- real (cosine-phase) input signal

*aim* -- imaginary (sine-phase) input signal

*am* -- amplitude modulation envelope

*af* -- frequency modulation envelope

*fmanal* takes a signal in quadrature (real, imag) and outputs the estimate AM and FM signals. The former is the modulus of each input sample pair ( $|re + j*im|$ ) and the latter the derivative of  $\arg(re + j*im)$ . Each sample of the output is the estimated instantaneous amplitude and frequency of the input signal.

## Example

Here is an example of the fmanal opcode. It uses the file *fmanal.csd* [examples/fmanal.csd].

### Example 320. Example of the fmanal opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

instr 1

asig oscili p4, p5
a1,a2 hilbert2 asig,1024,256
am,afm fmanal a1,a2
ktrig metro 2
printf "AM=%.3f FM=%.1f\n",ktrig,k(am),k(afm)
outs a1, a2
```

```
endin
```

```
</CsInstruments>  
<CsScore>  
i1 0 10 0.5 440  
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
2017

# fmax

fmax — Maximum value function.

## Description

Returns the maximum of its two arguments.

## Syntax

```
ires[] fmax iarg1[], iarg2[]  
kres[] fmax karg1[], karg2[]  
ires[] fmax iarg1[], iarg2  
kres[] fmax karg[], karg2
```

## Initialization

*iarg1*]/2, *iarg2* -- the operands.

## Performance

*karg1*]/2, *karg2* -- the operands

## Examples

Here is an example of the fmax opcode. It uses the file *fmax.csd* [examples/fmax.csd].

### Example 321. Example of the fmax opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
</CsOptions>  
<CsInstruments>  
  
instr 1  
  
iArg1[] fillarray 1,8,3  
iArg2[] fillarray 4,5,6  
iRes[] fmax iArg1,iArg2  
ik init 0  
  
while ik < lenarray(iRes) do  
  print iRes[ik]  
  ik += 1  
od  
  
endin
```

```
</CsInstruments>  
<CsScore>  
i1 0 0  
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
2017

# fmb3

fmb3 — Uses FM synthesis to create a Hammond B3 organ sound.

## Description

Uses FM synthesis to create a Hammond B3 organ sound. It comes from a family of FM sounds, all using 4 basic oscillators and various architectures, as used in the TX81Z synthesizer.

## Syntax

```
ares fmb3 kamp, kfreq, kc1, kc2, kvdepth, kvrate[, ifn1, ifn2, ifn3, \  
      ifn4, ivfn]
```

## Initialization

*fmb3* takes 5 tables for initialization. The first 4 are the basic inputs and the last is the low frequency oscillator (LFO) used for vibrato. The last table should usually be a sine wave. These all default to a sine wave table.

The initial waves should be:

- *ifn1* -- sine wave
- *ifn2* -- sine wave
- *ifn3* -- sine wave
- *ifn4* -- sine wave

## Performance

*kamp* -- Amplitude of note.

*kfreq* -- Frequency of note played.

*kc1*, *kc2* -- Controls for the synthesizer:

- *kc1* -- Total mod index
- *kc2* -- Crossfade of two modulators
- *Algorithm* -- 4

*kvdepth* -- Vibrator depth

*kvrate* -- Vibrator rate

## Examples

Here is an example of the fmb3 opcode. It uses the file *fmb3.csd* [examples/fmb3.csd].

### Example 322. Example of the fmb3 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o fmb3.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kfreq = 220
kc1 = p4
kc2 = p5
kvrata = 6

kvdpth line 0, p3, p6
asig fmb3 .4, kfreq, kc1, kc2, kvdpth, kvrate
outs asig, asig

endin
</CsInstruments>
<CsScore>
;sine wave.
f 1 0 32768 10 1

i 1 0 2 5 5 0.1
i 1 3 2 .5 .5 0.01
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*fmbell, fmmetal, fmpercfl, fmrhode, fmwurlie*

## Credits

Author: John ffitich (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

# fmbell

fmbell — Uses FM synthesis to create a tublar bell sound.

## Description

Uses FM synthesis to create a tublar bell sound. It comes from a family of FM sounds, all using 4 basic oscillators and various architectures, as used in the TX81Z synthesizer.

## Syntax

```
ares fmbell kamp, kfreq, kc1, kc2, kvdepth, kvrate[, ifn1, ifn2, ifn3, \  
ifn4, ivfn, isus]
```

## Initialization

All these opcodes take 5 tables for initialization. The first 4 are the basic inputs and the last is the low frequency oscillator (LFO) used for vibrato. These all default to a sawwave table.

The initial waves should be:

- *ifn1* -- sine wave
- *ifn2* -- sine wave
- *ifn3* -- sine wave
- *ifn4* -- sine wave

The optional argument *isus* controls how long the sound lasts, or how quickly it decays. It defaults to 4.

## Performance

*kamp* -- Amplitude of note.

*kfreq* -- Frequency of note played.

*kc1*, *kc2* -- Controls for the synthesizer:

- *kc1* -- Mod index 1
- *kc2* -- Crossfade of two outputs
- *Algorithm* -- 5

*kvdepth* -- Vibrator depth

*kvrate* -- Vibrator rate

## Examples

Here is an example of the fmbell opcode. It uses the file *fmbell.csd* [examples/fmbell.csd].

**Example 323. Example of the fmbell opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o fmbell.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kamp = p4
kfreq = 880
kc1 = p5
kc2 = p6
kvdepth = 0.005
kvrate = 6

asig fmbell kamp, kfreq, kc1, kc2, kvdepth, kvrate
outs asig, asig
endin

instr 2

kamp = p4
kfreq = 880
kc1 = p5
kc2 = p6
kvdepth = 0.005
kvrate = 6

asig fmbell kamp, kfreq, kc1, kc2, kvdepth, kvrate, 1, 1, 1, 1, 1, p7
outs asig, asig
endin

</CsInstruments>
<CsScore>
; sine wave.
f 1 0 32768 10 1

i 1 0 3 .2 5 5
i 1 + 4 .3 .5 1
s
i 2 0 12 .2 5 5 16
i 2 + 12 .3 .5 1 12

e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*fmb3, fmmetal, fmpercfl, fmrhode, fmwurlie*

## Credits

Author: John ffitich (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

Optional argument new in 5.16

More optional argument new in 6.0

# fmin

fmin — Minimum value function.

## Description

Returns the minimum of its two arguments.

## Syntax

```
ires[] fmin iarg1[], iarg2[]  
kres[] fmin karg1[], karg2[]  
ires[] fmin iarg1[], iarg2  
kres[] fmin karg[], karg2
```

## Initialization

*iarg1*]/2, *iarg2* -- the operands.

## Performance

*karg1*]/2, *karg2* -- the operands

## Examples

Here is an example of the fmin opcode. It uses the file *fmin.csd* [examples/fmin.csd].

### Example 324. Example of the fmin opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
</CsOptions>  
<CsInstruments>  
  
instr 1  
  
iArg1[] fillarray 1,8,3  
iArg2[] fillarray 4,5,6  
iRes[] fmin iArg1,iArg2  
ik init 0  
  
while ik < lenarray(iRes) do  
  print iRes[ik]  
  ik += 1  
od  
  
endin
```

```
</CsInstruments>  
<CsScore>  
i1 0 0  
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
2017

# fmmetal

fmmetal — Uses FM synthesis to create a “Heavy Metal” sound.

## Description

Uses FM synthesis to create a “Heavy Metal” sound. It comes from a family of FM sounds, all using 4 basic oscillators and various architectures, as used in the TX81Z synthesizer.

## Syntax

```
ares fmmetal kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, \  
      ifn4, ivfn
```

## Initialization

All these opcodes take 5 tables for initialization. The first 4 are the basic inputs and the last is the low frequency oscillator (LFO) used for vibrato. The last table should usually be a sine wave.

The initial waves should be:

- *ifn1* -- sine wave
- *ifn2* -- *twopeaks.aiff* [examples/twopeaks.aiff]
- *ifn3* -- *twopeaks.aiff* [examples/twopeaks.aiff]
- *ifn4* -- sine wave



### Note

The file “twopeaks.aiff” is also available at <ftp://ftp.cs.bath.ac.uk/pub/dream/documentation/sounds/modelling/>.

## Performance

*kamp* -- Amplitude of note.

*kfreq* -- Frequency of note played.

*kc1*, *kc2* -- Controls for the synthesizer:

- *kc1* -- Total mod index
- *kc2* -- Crossfade of two modulators
- *Algorithm* -- 3

*kvdepth* -- Vibrator depth

*kvrate* -- Vibrator rate

## Examples

Here is an example of the fmmetal opcode. It uses the file *fmmetal.csd* [examples/fmmetal.csd], and *twopeaks.aiff* [examples/twopeaks.aiff].

**Example 325. Example of the fmmetal opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o fmmetal.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kfreq = 440
kvdepth = 0
kvrates = 0
ifn1 = 1
ifn2 = 2
ifn3 = 2
ifn4 = 1
ivfn = 1
kc2 = p5

kc1 line p4, p3, 1
asig fmmetal .5, kfreq, kc1, kc2, kvdepth, kvrates, ifn1, ifn2, ifn3, ifn4, ivfn
outs asig, asig

endin
</CsInstruments>
<CsScore>
; sine wave.
f 1 0 32768 10 1
; the "twopeaks.aiff" audio file.
f 2 0 256 1 "twopeaks.aiff" 0 0 0

i 1 0 4 6 5
i 1 5 4 .2 10
e
</CsScore>
</CsoundSynthesizer>

```

**See Also**

*fmb3, fmbell, fmpercfl, fmrhode, fmwurlie*

**Credits**

Author: John ffitich (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

# fmod

fmod — Compute the floating point remainder operation.

## Description

Computes the remainder of the division of its first argument by the second.

## Syntax

```
ires[] fmod iarg1[], iarg2[]
kres[] fmod karg1[], karg2[]
ires[] fmod iarg1[], iarg2
kres[] fmod karg[], karg2
```

## Initialization

*iarg1*]/2, *iarg2* -- the operands.

## Performance

*karg1*]/2, *karg2* -- the operands

## Examples

Here is an example of the fmod opcode. It uses the file *fmod.csd* [examples/fmod.csd].

### Example 326. Example of the fmod opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsoundOptions>
</CsoundOptions>
<CsoundInstruments>

instr 1

iArg1[] fillarray 10,8,22
iArg2[] fillarray 4,5,6
iRes[] fmod iArg1,iArg2
ik init 0

while ik < lenarray(iRes) do
  print iRes[ik]
  ik += 1
od

endin
```

```
</CsInstruments>  
<CsScore>  
i1 0 0  
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
2017

# fmpercfl

fmpercfl — Uses FM synthesis to create a percussive flute sound.

## Description

Uses FM synthesis to create a percussive flute sound. It comes from a family of FM sounds, all using 4 basic oscillators and various architectures, as used in the TX81Z synthesizer.

## Syntax

```
ares fmpercfl kamp, kfreq, kc1, kc2, kvdepth, kvrate[, ifn1, ifn2, \  
    ifn3, ifn4, ivfn]
```

## Initialization

All these opcodes take 5 tables for initialization. The first 4 are the basic inputs and the last is the low frequency oscillator (LFO) used for vibrato. These all default to a sine wave table.

The initial waves should be:

- *ifn1* -- sine wave
- *ifn2* -- sine wave
- *ifn3* -- sine wave
- *ifn4* -- sine wave

## Performance

*kamp* -- Amplitude of note.

*kfreq* -- Frequency of note played.

*kc1, kc2* -- Controls for the synthesizer:

- *kc1* -- Total mod index
- *kc2* -- Crossfade of two modulators
- *Algorithm* -- 4

*kvdepth* -- Vibrator depth

*kvrate* -- Vibrator rate

## Examples

Here is an example of the fmpercfl opcode. It uses the file *fmpercfl.csd* [examples/fmpercfl.csd].

### Example 327. Example of the `fmpercfl` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o fmpercfl.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kfreq = 220
kc1 = 5
kvdepth = .01
kvrate = 6

kc2 line 5, p3, p4
asig fmpercfl .5, kfreq, kc1, kc2, kvdepth, kvrate
outs asig, asig
endin

</CsInstruments>
<CsScore>
; sine wave.
f 1 0 32768 10 1

i 1 0 4 5
i 1 5 8 .1

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*fmb3, fmbell, fmmetal, fmrhode, fmwurlie*

## Credits

Author: John ffitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

# fmrhode

fmrhode — Uses FM synthesis to create a Fender Rhodes electric piano sound.

## Description

Uses FM synthesis to create a Fender Rhodes electric piano sound. It comes from a family of FM sounds, all using 4 basic oscillators and various architectures, as used in the TX81Z synthesizer.

## Syntax

```
ares fmrhode kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, \  
      ifn3, ifn4, ivfn
```

## Initialization

All these opcodes take 5 tables for initialization. The first 4 are the basic inputs and the last is the low frequency oscillator (LFO) used for vibrato. The last table should usually be a sine wave.

The initial waves should be:

- *ifn1* -- sine wave
- *ifn2* -- sine wave
- *ifn3* -- sine wave
- *ifn4* -- *fwavblnk.aiff* [examples/fwavblnk.aiff]



### Note

The file “fwavblnk.aiff” is also available at <ftp://ftp.cs.bath.ac.uk/pub/dream/documentation/sounds/modelling/>.

## Performance

*kamp* -- Amplitude of note.

*kfreq* -- Frequency of note played.

*kc1*, *kc2* -- Controls for the synthesizer:

- *kc1* -- Mod index 1
- *kc2* -- Crossfade of two outputs
- *Algorithm* -- 5

*kvdepth* -- Vibrator depth

*kvrate* -- Vibrator rate

## Examples

Here is an example of the fmrhode opcode. It uses the file *fmrhode.csd* [examples/fmrhode.csd], and *fwavblnk.aiff* [examples/fwavblnk.aiff].

### Example 328. Example of the `fmrhode` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o fmrhode.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kfreq = 220
kc1 = p4
kc2 = p5
kvdepth = 0.01
kvrates = 3
ifn1 = 1
ifn2 = 1
ifn3 = 1
ifn4 = 2
ivfn = 1

asig fmrhode .5, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn
outs asig, asig

endin
</CsInstruments>
<CsScore>
; sine wave.
f 1 0 32768 10 1
; audio file.
f 2 0 256 1 "fwavblnk.aiff" 0 0 0

i 1 0 3 6 0
i 1 + . 6 3
i 1 + . 20 0
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*fmb3, fmbell, fmmetal, fnpercfl, fmwurlie*

## Credits

Author: John ffitich (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

# fmvoice

fmvoice — FM Singing Voice Synthesis

## Description

FM Singing Voice Synthesis

## Syntax

```
ares fmvoice kamp, kfreq, kvowel, ktilt, kvibamt, kvibrate[, ifn1, \  
    ifn2, ifn3, ifn4, ivibfn]
```

## Initialization

*ifn1, ifn2, ifn3, ifn4, ivibfn* -- Tables, usually of sinewaves. The last is for vibrato

## Performance

*kamp* -- Amplitude of note.

*kfreq* -- Frequency of note played.

*kvowel* -- the vowel being sung, in the range 0-64

*ktilt* -- the spectral tilt of the sound in the range 0 to 99

*kvibamt* -- Depth of vibrato

*kvibrate* -- Rate of vibrato

## Examples

Here is an example of the fmvoice opcode. It uses the file *fmvoice.csd* [examples/fmvoice.csd].

### Example 329. Example of the fmvoice opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
-odac      ;;realtime audio out  
;-iadc     ;;uncomment -iadc if realtime audio input is needed too  
; For Non-realtime ouput leave only the line below:  
; -o fmvoice.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>  
  
sr = 44100  
ksmps = 32  
nchnls = 2
```

```
Odbfs = 1

instr 1

kfreq = 110
kvowel = p4 ; p4 = vowel (0 - 64)
ktilt = p5
kvibamt = 0.005
kvibrate = 6

asig fmvoice .5, kfreq, kvowel, ktilt, kvibamt, kvibrate
outs asig, asig

endin
</CsInstruments>
<CsScore>
; sine wave.
f 1 0 16384 10 1

i 1 0 1 1 0 ; tilt=0
i 1 1 1 > .
i 1 2 1 > .
i 1 3 1 > .
i 1 4 1 > .
i 1 5 1 > .
i 1 6 1 > .
i 1 7 1 12 .

i 1 10 1 1 90 ; tilt=90
i 1 11 1 > .
i 1 12 1 > .
i 1 13 1 > .
i 1 14 1 > .
i 1 15 1 > .
i 1 16 1 > .
i 1 17 1 12 .

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitich (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

# fmwurlie

fmwurlie — Uses FM synthesis to create a Wurlitzer electric piano sound.

## Description

Uses FM synthesis to create a Wurlitzer electric piano sound. It comes from a family of FM sounds, all using 4 basic oscillators and various architectures, as used in the TX81Z synthesizer.

## Syntax

```
ares fmwurlie kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, \  
      ifn4, ivfn
```

## Initialization

All these opcodes take 5 tables for initialization. The first 4 are the basic inputs and the last is the low frequency oscillator (LFO) used for vibrato. The last table should usually be a sine wave.

The initial waves should be:

- *ifn1* -- sine wave
- *ifn2* -- sine wave
- *ifn3* -- sine wave
- *ifn4* -- *fwavblnk.aiff* [examples/fwavblnk.aiff]



### Note

The file “fwavblnk.aiff” is also available at <ftp://ftp.cs.bath.ac.uk/pub/dream/documentation/sounds/modelling/>.

## Performance

*kamp* -- Amplitude of note.

*kfreq* -- Frequency of note played.

*kc1*, *kc2* -- Controls for the synthesizer:

- *kc1* -- Mod index 1
- *kc2* -- Crossfade of two outputs
- *Algorithm* -- 5

*kvdepth* -- Vibrator depth

*kvrate* -- Vibrator rate

## Examples

Here is an example of the fmwurlie opcode. It uses the file *fmwurlie.csd* [examples/fmwurlie.csd], and *fwavblnk.aiff* [examples/fwavblnk.aiff].

### Example 330. Example of the `fmwurlie` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o fmwurlie.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kfreq = 440
kc1 = p4
kc2 = 1
kvdepth = 0.05
kvrates = 6
ifn1 = 1
ifn2 = 1
ifn3 = 1
ifn4 = 2
ivfn = 1

asig fmwurlie .5, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn
outs asig, asig

endin
</CsInstruments>
<CsScore>
; sine wave
f 1 0 32768 10 1
; audio file
f 2 0 256 1 "fwavblnk.aiff" 0 0 0

i 1 0 3 6
i 1 + 3 30
i 1 + 2 60
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*fmb3, fmbell, fmmetal, fnpercfl, fmrhode*

## Credits

Author: John ffitich (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

# fof

fof — Produces sinusoid bursts useful for formant and granular synthesis.

## Description

Audio output is a succession of sinusoid bursts initiated at frequency *xfund* with a spectral peak at *xform*. For *xfund* above 25 Hz these bursts produce a speech-like formant with spectral characteristics determined by the k-input parameters. For lower fundamentals this generator provides a special form of granular synthesis.

## Syntax

```
ares fof xamp, xfund, xform, koct, kband, kris, kdur, kdec, iolaps, \  
      ifna, ifnb, itotdur [, iphs] [, ifmode] [, iskip]
```

## Initialization

*iolaps* -- number of preallocated spaces needed to hold overlapping burst data. Overlaps are frequency dependent, and the space required depends on the maximum value of *xfund* \* *kdur*. Can be over-estimated at no computation cost. Uses less than 50 bytes of memory per *iolap*.

*ifna*, *ifnb* -- table numbers of two stored functions. The first is a sine table for sineburst synthesis (size of at least 4096 recommended). The second is a rise shape, used forwards and backwards to shape the sineburst rise and decay; this may be linear (*GEN07*) or perhaps a sigmoid (*GEN19*).

*itotdur* -- total time during which this *fof* will be active. Normally set to p3. No new sineburst is created if it cannot complete its *kdur* within the remaining *itotdur*.

*iphs* (optional, default=0) -- initial phase of the fundamental, expressed as a fraction of a cycle (0 to 1). The default value is 0.

*ifmode* (optional, default=0) -- formant frequency mode. If zero, each sineburst keeps the *xform* frequency it was launched with. If non-zero, each is influenced by *xform* continuously. The default value is 0.

*iskip* (optional, default=0) -- If non-zero, skip initialisation (allows legato use).

## Performance

*xamp* -- peak amplitude of each sineburst, observed at the true end of its rise pattern. The rise may exceed this value given a large bandwidth (say,  $Q < 10$ ) and/or when the bursts are overlapping.

*xfund* -- the fundamental frequency (in Hertz) of the impulses that create new sinebursts.

*xform* -- the formant frequency, i.e. freq of the sinusoid burst induced by each *xfund* impulse. This frequency can be fixed for each burst or can vary continuously (see *ifmode*).

*koct* -- octavation index, normally zero. If greater than zero, lowers the effective *xfund* frequency by attenuating odd-numbered sinebursts. Whole numbers are full octaves, fractions transitional.

*kband* -- the formant bandwidth (at -6dB), expressed in Hz. The bandwidth determines the rate of exponential decay throughout the sineburst, before the enveloping described below is applied.

*kris*, *kdur*, *kdec* -- rise, overall duration, and decay times (in seconds) of the sinusoid burst. These values apply an enveloped duration to each burst, in similar fashion to a Csound *linen* generator but with rise and decay shapes derived from the *ifnb* input. *kris* inversely determines the skirtwidth (at -40 dB) of the induced formant region. *kdur* affects the density of sineburst overlaps, and thus the speed of computation. Typical values for vocal imitation are .003,.02,.007.

Csound's *fof* generator is loosely based on Michael Clarke's C-coding of IRCAM's *CHANT* program (Xavier Rodet et al.). Each *fof* produces a single formant, and the output of four or more of these can be summed to produce a rich vocal imitation. *fof* synthesis is a special form of granular synthesis, and this implementation aids transformation between vocal imitation and granular textures. Computation speed depends on *kdur*, *xfund*, and the density of any overlaps.

## Examples

Here is an example of the *fof* opcode. It uses the file *fof.csd* [examples/fof.csd].

### Example 331. Example of the *fof* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o fof.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

instr 1
; Combine five formants together to create
; a transformation from an alto-"a" sound
; to an alto-"i" sound.
; Values common to all of the formants.
kfund init 261.659
kocf init 0
kris init 0.003
kdur init 0.02
kdec init 0.007
iolaps = 100
ifna = 1
ifnb = 2
itotdur = p3

; First formant.
k1amp = ampdb(0)
k1form line 800, p3, 350
k1band line 80, p3, 50

; Second formant.
k2amp line ampdb(-4), p3, ampdb(-20)
k2form line 1150, p3, 1700
k2band line 90, p3, 100

; Third formant.
```

```
k3amp line ampdb(-20), p3, ampdb(-30)
k3form line 2800, p3, 2700
k3band init 120

; Fourth formant.
k4amp init ampdb(-36)
k4form line 3500, p3, 3700
k4band line 130, p3, 150

; Fifth formant.
k5amp init ampdb(-60)
k5form init 4950
k5band line 140, p3, 200

a1 fof k1amp, kfund, k1form, koct, k1band, kris, \
    kdur, kdec, iolaps, ifna, ifnb, itotdur
a2 fof k2amp, kfund, k2form, koct, k2band, kris, \
    kdur, kdec, iolaps, ifna, ifnb, itotdur
a3 fof k3amp, kfund, k3form, koct, k3band, kris, \
    kdur, kdec, iolaps, ifna, ifnb, itotdur
a4 fof k4amp, kfund, k4form, koct, k4band, kris, \
    kdur, kdec, iolaps, ifna, ifnb, itotdur
a5 fof k5amp, kfund, k5form, koct, k5band, kris, \
    kdur, kdec, iolaps, ifna, ifnb, itotdur

; Combine all of the formants together
asig sum (a1+a2+a3+a4+a5) * 13000
outs asig, asig

endin
</CsInstruments>
<CsScore>
; sine wave
f 1 0 4096 10 1
; sigmoid wave
f 2 0 1024 19 0.5 0.5 270 0.5

i 1 0 1
i 1 2 5 ; same but slower
e
</CsScore>
</CsSoundSynthesizer>
```

The formant values for the alto-"a" sound were taken from the *Formant Values Appendix*.

## See Also

*fof2, Formant Values Appendix*

## Credits

Added in version 1 (1990)

# fof2

fof2 — Produces sinusoid bursts including k-rate incremental indexing with each successive burst.

## Description

Audio output is a succession of sinusoid bursts initiated at frequency *xfund* with a spectral peak at *xform*. For *xfund* above 25 Hz these bursts produce a speech-like formant with spectral characteristics determined by the k-input parameters. For lower fundamentals this generator provides a special form of granular synthesis.

*fof2* implements k-rate incremental indexing into *ifna* function with each successive burst.

## Syntax

```
ares fof2 xamp, xfund, xform, koct, kband, kris, kdur, kdec, iolaps, \  
      ifna, ifnb, itotdur, kphs, kgliss [, iskip]
```

## Initialization

*iolaps* -- number of preallocated spaces needed to hold overlapping burst data. Overlaps are frequency dependent, and the space required depends on the maximum value of *xfund* \* *kdur*. Can be over-estimated at no computation cost. Uses less than 50 bytes of memory per *iolap*.

*ifna*, *ifnb* -- table numbers of two stored functions. The first is a sine table for sineburst synthesis (size of at least 4096 recommended). The second is a rise shape, used forwards and backwards to shape the sineburst rise and decay; this may be linear (*GEN07*) or perhaps a sigmoid (*GEN19*).

*itotdur* -- total time during which this *fof* will be active. Normally set to p3. No new sineburst is created if it cannot complete its *kdur* within the remaining *itotdur*.

*iskip* (optional, default=0) -- If non-zero, skip initialization (allows legato use).

## Performance

*xamp* -- peak amplitude of each sineburst, observed at the true end of its rise pattern. The rise may exceed this value given a large bandwidth (say,  $Q < 10$ ) and/or when the bursts are overlapping.

*xfund* -- the fundamental frequency (in Hertz) of the impulses that create new sinebursts.

*xform* -- the formant frequency, i.e. freq of the sinusoid burst induced by each *xfund* impulse. This frequency can be fixed for each burst or can vary continuously (see *ifmode*).

*koct* -- octavation index, normally zero. If greater than zero, lowers the effective *xfund* frequency by attenuating odd-numbered sinebursts. Whole numbers are full octaves, fractions transitional.

*kband* -- the formant bandwidth (at -6dB), expressed in Hz. The bandwidth determines the rate of exponential decay throughout the sineburst, before the enveloping described below is applied.

*kris*, *kdur*, *kdec* -- rise, overall duration, and decay times (in seconds) of the sinusoid burst. These values apply an enveloped duration to each burst, in similar fashion to a Csound *linen* generator but with rise and decay shapes derived from the *ifnb* input. *kris* inversely determines the skirtwidth (at -40 dB) of the induced formant region. *kdur* affects the density of sineburst overlaps, and thus the speed of computation. Typical values for vocal imitation are .003,.02,.007.

*kphs* -- allows k-rate indexing of function table *ifna* with each successive burst, making it suitable for time-warping applications. Values of *kphs* are normalized from 0 to 1, 1 being the end of the function table *ifna*.

*kgliss* -- sets the end pitch of each grain relative to the initial pitch, in octaves. Thus *kgliss* = 2 means that the grain ends two octaves above its initial pitch, while *kgliss* = -3/4 has the grain ending a major sixth below. Each 1/12 added to *kgliss* raises the ending frequency one half-step. If you want no glissando, set *kgliss* to 0.

Csound's *fof* generator is loosely based on Michael Clarke's C-coding of IRCAM's *CHANT* program (Xavier Rodet et al.). Each *fof* produces a single formant, and the output of four or more of these can be summed to produce a rich vocal imitation. *fof* synthesis is a special form of granular synthesis, and this implementation aids transformation between vocal imitation and granular textures. Computation speed depends on *kdur*, *xfund*, and the density of any overlaps.



### Note

The ending frequency of any grain is equal to  $kform * (2 ^ kgliss)$ , so setting *kgliss* too high may result in aliasing. For example, *kform* = 3000 and *kgliss* = 3 places the ending frequency over the Nyquist if *sr* = 44100. It is a good idea to scale *kgliss* accordingly.

## Examples

Here is an example of the *fof2* opcode. It uses the file *fof2.csd* [examples/fof2.csd].

### Example 332. Example of the *fof2* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o fof2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 128
nchnls = 2

;By Andres Cabrera 2007

instr 1          ;table-lookup vocal synthesis

kris init p12
kdur init p13
kdec init p14

iolaps init p15

ifna init 1 ; Sine wave
ifnb init 2 ; Straight line rise shape

itotdur init p3

kphs init 0 ; No phase modulation (constant kphs)
```

```

kfund line p4, p3, p5
kform line p6, p3, p7
koct line p8, p3, p9
kband line p10, p3, p11
kgliss line p16, p3, p17

kenv linen 5000, 0.03, p3, 0.03 ;to avoid clicking

aout fof2 kenv, kfund, kform, koct, kband, kris, kdur, kdec, iolaps, \
      ifna, ifnb, itotdur, kphs, kgliss

      outs aout, aout
      endin

</CsInstruments>
<CsScore>
f1 0 8192 10 1
f2 0 4096 7 0 4096 1

;          kfund1 kfund2 kform1 kform2 koct1 koct2 kband1 kband2 kris kdur kdec iolaps kg
i1 0 4 220 220 510 510 0 0 30 30 0.01 0.03 0.01 20
i1 + . 220 220 510 910 0 0 30 30 0.01 0.03 0.01 20
i1 + . 220 220 510 510 0 0 100 30 0.01 0.03 0.01 20
i1 + . 220 220 510 510 0 1 30 30 0.01 0.03 0.01 20
i1 + . 220 220 510 510 0 0 30 30 0.01 0.03 0.01 20
i1 + . 220 220 510 510 0 0 30 30 0.01 0.05 0.01 100
i1 + . 220 440 510 510 0 0 30 30 0.01 0.05 0.01 100

e

</CsScore>
</CsoundSynthesizer>

```

Here is another example of the fof2 opcode, which generates vowel tones using formants generated by fof2 coinciding with values from the *Formant Values* appendix. It uses the file *fof2-2.csd* [examples/fof2-2.csd].

### Example 333. Example of the fof2 opcode to produce vowel sounds.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out Audio in
-odac -iadc ;;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o fof2.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 128
nchnls = 2

; Example by Chuckk Hubbard 2007

instr 1 ;table-lookup vocal synthesis

iolaps = 120
ifna = 1 ;f1 - sine wave
ifnb = 2 ;f2 - linear rise shape
itotdur = p3
iamp = p4 * 0dbfs
ifreq1 = p5 ;starting frequency

```

```

ifreq2    =    p6            ;ending frequency

kamp      linseg    0, .003, iamp, itotdur-.007, iamp, .003, 0, .001, 0
kfund     expseg    ifreq1, itotdur, ifreq2
koct      init      0
kris      init      .003
kdur      init      .02
kdec      init      .007
kphs      init      0
kgliss    init      0

iforma    =    p7            ;starting spectrum
iformb    =    p8            ;ending spectrum

iform1a   tab_i     0, iforma      ;read values of 5 formants for 1st spectrum
iform2a   tab_i     1, iforma
iform3a   tab_i     2, iforma
iform4a   tab_i     3, iforma
iform5a   tab_i     4, iforma
idb1a     tab_i     5, iforma      ;read decibel levels for same 5 formants
idb2a     tab_i     6, iforma
idb3a     tab_i     7, iforma
idb4a     tab_i     8, iforma
idb5a     tab_i     9, iforma
iband1a   tab_i     10, iforma     ;read bandwidths for same 5 formants
iband2a   tab_i     11, iforma
iband3a   tab_i     12, iforma
iband4a   tab_i     13, iforma
iband5a   tab_i     14, iforma
iamp1a    =    ampdb(idb1a)        ;convert db to linear multipliers
iamp2a    =    ampdb(idb2a)
iamp3a    =    ampdb(idb3a)
iamp4a    =    ampdb(idb4a)
iamp5a    =    ampdb(idb5a)

iform1b   tab_i     0, iformb     ;values of 5 formants for 2nd spectrum
iform2b   tab_i     1, iformb
iform3b   tab_i     2, iformb
iform4b   tab_i     3, iformb
iform5b   tab_i     4, iformb
idb1b     tab_i     5, iformb     ;decibel levels for 2nd set of formants
idb2b     tab_i     6, iformb
idb3b     tab_i     7, iformb
idb4b     tab_i     8, iformb
idb5b     tab_i     9, iformb
iband1b   tab_i     10, iformb    ;bandwidths for 2nd set of formants
iband2b   tab_i     11, iformb
iband3b   tab_i     12, iformb
iband4b   tab_i     13, iformb
iband5b   tab_i     14, iformb
iamp1b    =    ampdb(idb1b)        ;convert db to linear multipliers
iamp2b    =    ampdb(idb2b)
iamp3b    =    ampdb(idb3b)
iamp4b    =    ampdb(idb4b)
iamp5b    =    ampdb(idb5b)

kform1    line      iform1a, itotdur, iform1b      ;transition between formants
kform2    line      iform2a, itotdur, iform2b
kform3    line      iform3a, itotdur, iform3b
kform4    line      iform4a, itotdur, iform4b
kform5    line      iform5a, itotdur, iform5b
kband1    line      iband1a, itotdur, iband1b      ;transition of bandwidths
kband2    line      iband2a, itotdur, iband2b
kband3    line      iband3a, itotdur, iband3b
kband4    line      iband4a, itotdur, iband4b
kband5    line      iband5a, itotdur, iband5b
kamp1     line      iamp1a, itotdur, iamp1b        ;transition of amplitudes of formants

```

## Orchestra Opcodes and Operators

```

kamp2  line  iamp2a, itotdur, iamp2b
kamp3  line  iamp3a, itotdur, iamp3b
kamp4  line  iamp4a, itotdur, iamp4b
kamp5  line  iamp5a, itotdur, iamp5b

;5 formants for each spectrum
a1  fof2  kamp1, kfund, kform1, koct, kband1, kris, kdur, kdec, iolaps, ifna, ifnb, itotdur, kphs,
a2  fof2  kamp2, kfund, kform2, koct, kband2, kris, kdur, kdec, iolaps, ifna, ifnb, itotdur, kphs,
a3  fof2  kamp3, kfund, kform3, koct, kband3, kris, kdur, kdec, iolaps, ifna, ifnb, itotdur, kphs,
a4  fof2  kamp4, kfund, kform4, koct, kband4, kris, kdur, kdec, iolaps, ifna, ifnb, itotdur, kphs,
a5  fof2  kamp5, kfund, kform5, koct, kband5, kris, kdur, kdec, iolaps, ifna, ifnb, itotdur, kphs,

aout   =   (a1+a2+a3+a4+a5) * kamp/5   ;sum and scale

aenv  linen 1, 0.05, p3, 0.05 ;to avoid clicking

      outs   aout*aenv, aout*aenv
      endin

</CsInstruments>
<CsScore>
f1 0 8192 10 1
f2 0 4096 7 0 4096 1

;*****
; tables of formant values adapted from MiscFormants.html
; 100's: soprano   200's: alto   300's: countertenor   400's: tenor   500's: bass
; -01: "a" sound  -02: "e" sound  -03: "i" sound  -04: "o" sound  -05: "u" sound
; p-5 through p-9: frequencies of 5 formants
; p-10 through p-14: decibel levels of 5 formants
; p-15 through p-19: bandwidths of 5 formants

;          formant frequencies          decibel levels          bandwidths
;soprano
f101 0 16 -2   800    1150    2900    3900    4950    0.001   -6    -32    -20    -50
f102 0 16 -2   350    2000    2800    3600    4950    0.001  -20   -15   -40   -56   6
f103 0 16 -2   270    2140    2950    3900    4950    0.001  -12   -26   -26   -44   60
f104 0 16 -2   450    800     2830    3800    4950    0.001  -11   -22   -22   -50   40
f105 0 16 -2   325    700     2700    3800    4950    0.001  -16   -35   -40   -60   50
;alto
f201 0 16 -2   800    1150    2800    3500    4950    0.001  -4    -20   -36   -60   8
f202 0 16 -2   400    1600    2700    3300    4950    0.001 -24   -30   -35   -60
f203 0 16 -2   350    1700    2700    3700    4950    0.001 -20   -30   -36   -60
f204 0 16 -2   450    800     2830    3500    4950    0.001  -9   -16   -28   -55
f205 0 16 -2   325    700     2530    3500    4950    0.001 -12   -30   -40   -64
;countertenor
f301 0 16 -2   660    1120    2750    3000    3350    0.001  -6   -23   -24   -38
f302 0 16 -2   440    1800    2700    3000    3300    0.001 -14   -18   -20   -20   70
f303 0 16 -2   270    1850    2900    3350    3590    0.001 -24   -24   -36   -36   40
f304 0 16 -2   430    820     2700    3000    3300    0.001 -10   -26   -22   -34   40
f305 0 16 -2   370    630     2750    3000    3400    0.001 -20   -23   -30   -34   40
;tenor
f401 0 16 -2   650    1080    2650    2900    3250    0.001  -6   -7    -8   -22   8
f402 0 16 -2   400    1700    2600    3200    3580    0.001 -14   -12   -14   -20   70
f403 0 16 -2   290    1870    2800    3250    3540    0.001 -15   -18   -20   -30   40
f404 0 16 -2   400    800     2600    2800    3000    0.001 -10   -12   -12   -26   70
f405 0 16 -2   350    600     2700    2900    3300    0.001 -20   -17   -14   -26   40
;bass
f501 0 16 -2   600    1040    2250    2450    2750    0.001  -7   -9   -9   -20   6
f502 0 16 -2   400    1620    2400    2800    3100    0.001 -12   -9   -12   -18
f503 0 16 -2   250    1750    2600    3050    3340    0.001 -30   -16   -22   -28
f504 0 16 -2   400    750     2400    2600    2900    0.001 -11   -21   -20   -40
f505 0 16 -2   350    600     2400    2675    2950    0.001 -20   -32   -28   -36
;*****

; start dur  amp  start freq  end freq  start formant  end formant
i1  0  1  .8  440 412.5 201 203

```

```

i1 + . .8 412.5 550 201 204
i1 + . .8 495 330 202 205

i1 + . .8 110 103.125 501 503
i1 + . .8 103.125 137.5 501 504
i1 + . .8 123.75 82.5 502 505

i1 7 . .4 440 412.5 201 203
i1 8 . .4 412.5 550 201 204
i1 9 . .4 495 330 202 205
i1 7 . .4 110 103.125 501 503
i1 8 . .4 103.125 137.5 501 504
i1 9 . .4 123.75 82.5 502 505
i1 + . .4 440 412.5 101 103
i1 + . .4 412.5 550 101 104
i1 + . .4 495 330 102 105
e

```

```

</CsScore>
</CsoundSynthesizer>

```

## See Also

*fof*

## Credits

Author: Rasmus Ekman  
*fof2* is a modification of *fof* by Rasmus Ekman

New in Csound 3.45

# fofilter

fofilter — Formant filter.

## Description

Fofilter generates a stream of overlapping sinewave grains, when fed with a pulse train. Each grain is the impulse response of a combination of two BP filters. The grains are defined by their attack time (determining the skirtwidth of the formant region at -60dB) and decay time (-6dB bandwidth). Overlapping will occur when  $1/\text{freq} < \text{decay}$ , but, unlike FOF, there is no upper limit on the number of overlaps. The original idea for this opcode came from J McCartney's formlet class in SuperCollider, but this is possibly implemented differently(?).

## Syntax

```
asig fofilter ain, xcf, xris, xdec[, istor]
```

## Initialization

*istor* --initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*asig* -- input signal.

*xcf* -- filter centre frequency

*xris* -- impulse response attack time (secs).

*xdec* -- impulse response decay time (secs).

## Examples

Here is an example of the fofilter opcode. It uses the file *fofilter.csd* [examples/fofilter.csd].

### Example 334. Example of the fofilter opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o fofilter.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kfe expseg 10, p3*0.9, 180, p3*0.1, 175
kenv linen .1, 0.05, p3, 0.05
asig buzz kenv, kfe, sr/(2*kfe), 1
afil fofilter asig, 900, 0.007, 0.04
outs afil, afil

endin
</CsInstruments>
<CsScore>
; sine wave
f 1 0 16384 10 1

i 1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
January 2005

New plugin in version 5

Audio rate parameters introduced in version 6.02

November 2013.

# fog

fog — Audio output is a succession of grains derived from data in a stored function table

## Description

Audio output is a succession of grains derived from data in a stored function table *ifna*. The local envelope of these grains and their timing is based on the model of *fof* synthesis and permits detailed control of the granular synthesis.

## Syntax

```
ares fog xamp, xdens, xtrans, aspd, koct, kband, kris, kdur, kdec, \  
        iolaps, ifna, ifnb, itotdur [, iphs] [, itmode] [, iskip]
```

## Initialization

*iolaps* -- number of pre-located spaces needed to hold overlapping grain data. Overlaps are density dependent, and the space required depends on the maximum value of  $xdens * kdur$ . Can be over-estimated at no computation cost. Uses less than 50 bytes of memory per *iolap*.

*ifna*, *ifnb* -- table numbers of two stored functions. The first is the data used for granulation, usually from a soundfile (*GEN01*). The second is a rise shape, used forwards and backwards to shape the grain rise and decay; this is normally a sigmoid (*GEN19*) but may be linear (*GEN05*).

*itotdur* -- total time during which this *fog* will be active. Normally set to p3. No new grain is created if it cannot complete its *kdur* within the remaining *itotdur*.

*iphs* (optional) -- initial phase of the fundamental, expressed as a fraction of a cycle (0 to 1). The default value is 0.

*itmode* (optional) -- transposition mode. If zero, each grain keeps the *xtrans* value it was launched with. If non-zero, each is influenced by *xtrans* continuously. The default value is 0.

*iskip* (optional, default=0) -- If non-zero, skip initialization (allows legato use).

## Performance

*xamp* -- amplitude factor. Amplitude is also dependent on the number of overlapping grains, the interaction of the rise shape (*ifnb*) and the exponential decay (*kband*), and the scaling of the grain waveform (*ifna*). The actual amplitude may therefore exceed *xamp*.

*xdens* -- density. The frequency of grains per second.

*xtrans* -- transposition factor. The rate at which data from the stored function table *ifna* is read within each grain. This has the effect of transposing the original material. A value of 1 produces the original pitch. Higher values transpose upwards, lower values downwards. Negative values result in the function table being read backwards.

*aspd* -- Starting index pointer. *aspd* is the normalized index (0 to 1) to table *ifna* that determines the movement of a pointer used as the starting point for reading data within each grain. (*xtrans* determines the rate at which data is read starting from this pointer.)

*koct* -- octavation index. The operation of this parameter is identical to that in *fof*.

*kband*, *kris*, *kdur*, *kdec* -- grain envelope shape. These parameters determine the exponential decay (*kband*), and the rise (*kris*), overall duration (*kdur*), and decay (*kdec*) times of the grain envelope. Their operation is identical to that of the local envelope parameters in *fof*.

## Examples

Here is an example of the fog opcode. It uses the file *fog.csd* [examples/fog.csd].

### Example 335. Example of the fog opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o fog.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

;p4 = transposition factor
;p5 = speed factor
;p6 = function table for grain data
i1 = sr/ftlen(1) ;scaling to reflect sample rate and table length
a1 phasor i1*p5 ;index for speed
asigl fog .5, 15, p4, a1, 1, 0, .01, .5, .01, 30, 1, 2, p3 ;left channel
asigr fog .4, 25, p4+.2, a1, 1, 0, .01, .5, .01, 30, 1, 2, p3, .5 ;right channel
outs asigl, asigr
endin

</CsInstruments>
<CsScore>
f 1 0 131072 1 "fox.wav" 0 0 0
f 2 0 1024 19 .5 .5 270 .5

i 1 0 10 .7 .1
i 1 + 4 1.2 2
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Michael Clark  
Huddersfield  
May 1997

New in version 3.46

The Csound fog generator is by Michael Clarke, extending his earlier work based on IRCAM's FOF algorithm.

Added notes by Rasmus Ekman on September 2002.

# fold

fold — Adds artificial foldover to an audio signal.

## Description

Adds artificial foldover to an audio signal.

## Syntax

```
ares fold asig, kincr
```

## Performance

*asig* -- input signal

*kincr* -- amount of foldover expressed in multiple of sampling rate. Must be  $\geq 1$

*fold* is an opcode which creates artificial foldover. For example, when *kincr* is equal to 1 with *sr*=44100, no foldover is added. When *kincr* is set to 2, the foldover is equivalent to a downsampling to 22050, when it is set to 4, to 11025 etc. Fractional values of *kincr* are possible, allowing a continuous variation of foldover amount. This can be used for a wide range of special effects.

## Examples

Here is an example of the fold opcode. It uses the file *fold.csd* [examples/fold.csd].

### Example 336. Example of the fold opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o fold.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

asig poscil3 .8, 400, 1 ,very clean sine
kincr line p4, p3, p5
asig fold asig, kincr
outs asig, asig

endin
```

```
</CsInstruments>
<CsScore>
; sine wave.
f 1 0 16384 10 1

i 1 0 4 2 2
i 1 5 4 5 5
i 1 10 4 10 10
i 1 15 4 1 100 ; Vary the fold-over amount from 1 to 100

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Gabriel Maldonado  
Italy  
1999

New in Csound version 3.56

# follow

follow — Envelope follower unit generator.

## Description

Envelope follower unit generator.

## Syntax

```
ares follow asig, idt
```

## Initialization

*idt* -- This is the period, in seconds, that the average amplitude of *asig* is reported. If the frequency of *asig* is low then *idt* must be large (more than half the period of *asig* )

## Performance

*asig* -- This is the signal from which to extract the envelope.

## Examples

Here is an example of the follow opcode. It uses the file *follow.csd* [examples/follow.csd], and *beats.wav* [examples/beats.wav].

### Example 337. Example of the follow opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o follow.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

asig soundin "beats.wav"
outs asig, asig

endin

instr 2 ;envelope follower
```

```
as  soundin "beats.wav"
as  = as*.7  ;reduce volume a bit
at  tone   as, 500 ;smooth estimated envelope
af  follow at, p4
asin poscil3 .5, 440, 1
; "beats.wav" provides amplitude for poscil
asig balance asin, af
    outs   asig, asig

endin
</CsInstruments>
<CsScore>
; sine wave.
f 1 0 32768 10 1

i 1 0 2
i 2 2 2 0.001 ;follow quickly
i 2 5 3 0.2   ;follow slowly
e
</CsScore>
</CsoundSynthesizer>
```

To avoid zipper noise, by discontinuities produced from complex envelope tracking, a lowpass filter could be used, to smooth the estimated envelope.

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

# follow2

follow2 — Another controllable envelope extractor.

## Description

A controllable envelope extractor using the algorithm attributed to Jean-Marc Jot.

## Syntax

```
ares follow2 asig, katt, krel
```

## Performance

*asig* -- the input signal whose envelope is followed

*katt* -- the attack rate (60dB attack time in seconds)

*krel* -- the decay rate (60dB decay time in seconds)

The output tracks the amplitude envelope of the input signal. The rate at which the output grows to follow the signal is controlled by the *katt*, and the rate at which it decreases in response to a lower amplitude, is controlled by the *krel*. This gives a smoother envelope than *follow*.

## Examples

Here is an example of the follow2 opcode. It uses the file *follow2.csd* [examples/follow2.csd], and *beats.wav* [examples/beats.wav].

### Example 338. Example of the follow2 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o follow2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

asig soundin "beats.wav"
outs asig, asig
endin
```

```
instr 2 ;using follow2

as soundin "beats.wav"
af follow2 as, p4, p5
ar rand 44100 ;noise
; "beats.wav" provides amplitude for noise
asig balance ar, af
    outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 2
i 2 2 2 0.001 0.01 ;quick attack & decay
i 2 5 2 0.1 0.5 ;slow attack & decay

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitch

The algorithm for the *follow2* is attributed to Jean-Marc Jot.

University of Bath, Codemist Ltd.

Bath, UK

February 2000

New in Csound version 4.03

Added notes by Rasmus Ekman on September 2002.

# foscil

foscil — A basic frequency modulated oscillator.

## Description

A basic frequency modulated oscillator.

## Syntax

```
ares foscil xamp, kcps, xcar, xmod, kndx, ifn [, iphs]
```

## Initialization

*ifn* -- function table number. Requires a wrap-around guard point.

*iphs* (optional, default=0) -- initial phase of waveform in table *ifn*, expressed as a fraction of a cycle (0 to 1). A negative value will cause phase initialization to be skipped. The default value is 0.

## Performance

*xamp* -- the amplitude of the output signal.

*kcps* -- a common denominator, in cycles per second, for the carrier and modulating frequencies.

*xcar* -- a factor that, when multiplied by the *kcps* parameter, gives the carrier frequency.

*xmod* -- a factor that, when multiplied by the *kcps* parameter, gives the modulating frequency.

*kndx* -- the modulation index.

*foscil* is a composite unit that effectively banks two *oscil* opcodes in the familiar Chowning FM setup, wherein the audio-rate output of one generator is used to modulate the frequency input of another (the “carrier”). Effective carrier frequency =  $kcps * xcar$ , and modulating frequency =  $kcps * xmod$ . For integral values of *xcar* and *xmod*, the perceived fundamental will be the minimum positive value of  $kcps * (xcar - n * xmod)$ ,  $n = 0, 1, 2, \dots$ . The input *kndx* is the index of modulation (usually time-varying and ranging 0 to 4 or so) which determines the spread of acoustic energy over the partial positions given by  $n = 0, 1, 2, \dots$ , etc. *ifn* should point to a stored sine wave. Previous to version 3.50, *xcar* and *xmod* could be k-rate only.

The actual formula used for this implementation of FM synthesis is  $xamp * \cos(2\pi * t * kcps * xcar + kndx * \sin(2\pi * t * kcps * xmod) - \pi)$ , assuming that the table is a sine wave.

## Examples

Here is an example of the *foscil* opcode. It uses the file *foscil.csd* [examples/foscil.csd].

### Example 339. Example of the foscil opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o foscil.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kcps = 440
kcar = 1
kmod = p4
kndx line 0, p3, 20 ;intensivy sidebands

asig foscil .5, kcps, kcar, kmod, kndx, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
; sine
f 1 0 16384 10 1

i 1 0 9 .01 ;vibrato
i 1 10 . 1
i 1 20 . 1.414 ;gong-ish
i 1 30 5 2.05 ;with "beat"
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

More information about frequency modulation on Wikipedia: [http://en.wikipedia.org/wiki/Frequency\\_modulation\\_synthesis](http://en.wikipedia.org/wiki/Frequency_modulation_synthesis)

# foscili

foscili — Basic frequency modulated oscillator with linear interpolation.

## Description

Basic frequency modulated oscillator with linear interpolation.

## Syntax

```
ares foscili xamp, kcps, xcar, xmod, kndx, ifn [, iphs]
```

## Initialization

*ifn* -- function table number. Requires a wrap-around guard point.

*iphs* (optional, default=0) -- initial phase of waveform in table *ifn*, expressed as a fraction of a cycle (0 to 1). A negative value will cause phase initialization to be skipped. The default value is 0.

## Performance

*xamp* -- the amplitude of the output signal.

*kcps* -- a common denominator, in cycles per second, for the carrier and modulating frequencies.

*xcar* -- a factor that, when multiplied by the *kcps* parameter, gives the carrier frequency.

*xmod* -- a factor that, when multiplied by the *kcps* parameter, gives the modulating frequency.

*kndx* -- the modulation index.

*foscili* differs from *foscil* in that the standard procedure of using a truncated phase as a sampling index is here replaced by a process that interpolates between two successive lookups. Interpolating generators will produce a noticeably cleaner output signal, but they may take as much as twice as long to run. Adequate accuracy can also be gained without the time cost of interpolation by using large stored function tables of 2K, 4K or 8K points if the space is available.

## Examples

Here is an example of the *foscili* opcode. It uses the file *foscili.csd* [examples/foscili.csd].

### Example 340. Example of the *foscili* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
```

```
; -o foscili.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kcps = 440
kcar = 1
kmod = p4
kndx line 0, p3, 20 ;intensivy sidebands

asig foscili .5, kcps, kcar, kmod, kndx, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
; sine
f 1 0 16384 10 1

i 1 0 9 .01 ;vibrato
i 1 10 . 1
i 1 20 . 1.414 ;gong-ish
i 1 30 5 2.05 ;with "beat"
e
</CsScore>
</CsSoundSynthesizer>
```

## See Also

More information about frequency modulation on Wikipedia: [http://en.wikipedia.org/wiki/Frequency\\_modulation\\_synthesis](http://en.wikipedia.org/wiki/Frequency_modulation_synthesis)

# fout

`fout` — Outputs a-rate signals to an arbitrary number of channels.

## Description

`fout` outputs  $N$  a-rate signals to a specified file of  $N$  channels.

## Syntax

```
fout ifilename, iformat, aout1 [, aout2, aout3, ..., aoutN]
```

```
fout ifilename, iformat, array[]
```

## Initialization

`ifilename` -- the output file's name (in double-quotes).

`iformat` -- a flag to choose output file format (note: Csound versions older than 5.0 may only support formats 0, 1, and 2):

- 0 - 32-bit floating point samples without header (binary PCM multichannel file)
- 1 - 16-bit integers without header (binary PCM multichannel file)
- 2 - 16-bit integers with a header. The header type depends on the render (-o) format. For example, if the user chooses the AIFF format (using the *-A flag*), the header format will be AIFF type.
- 3 - u-law samples with a header (see `iformat=2`).
- 4 - 16-bit integers with a header (see `iformat=2`).
- 5 - 32-bit integers with a header (see `iformat=2`).
- 6 - 32-bit floats with a header (see `iformat=2`).
- 7 - 8-bit unsigned integers with a header (see `iformat=2`).
- 8 - 24-bit integers with a header (see `iformat=2`).
- 9 - 64-bit floats with a header (see `iformat=2`).

In addition, Csound versions 5.0 and later allow for explicitly selecting a particular header type by specifying the format as `10 * fileType + sampleFormat`, where `fileType` may be 1 for WAV, 2 for AIFF, 3 for raw (headerless) files, and 4 for IRCAM; `sampleFormat` is one of the above values in the range 0 to 9, except sample format 0 is taken from the command line (-o), format 1 is 8-bit signed integers, and format 2 is a-law. So, for example, `iformat=25` means 32-bit integers with AIFF header.

## Performance

`aout1, ..., aoutN` -- signals to be written to the file. In the case of raw files, the expected range of audio signals is determined by the selected sample format; for sound files with a header like WAV and AIFF, the audio signals should be in the range -0dbfs to 0dbfs.

`fout` (file output) writes samples of audio signals to a file with any number of channels. Channel number depends by the number of `aoutN` variables (i.e. a mono signal with only an a-rate argument, a stereo signal

with two a-rate arguments etc.) Maximum number of channels is fixed to 64. Multiple *fout* opcodes can be present in the same instrument, referring to different files.

Notice that, unlike *out*, *outs* and *outq*, *fout* does not zero the audio variable so you must zero it after calling it. If polyphony is to be used, you can use *vincr* and *clear* opcodes for this task.

Notice that *fout* and *foutk* uses a string containing a file pathname. Whereas, with *fouti* and *foutir*, the target file can be only specified by means of a handle-number.



## Note

If you are using *fout* to generate an audio file for the global output of csound, you might want to use the *monitor* opcode, which can tap the output buffer, to avoid having to route

## Examples

Here is a simple example of the *fout* opcode. It uses the file *fout.csd* [examples/fout.csd].

### Example 341. Example of the *fout* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o fout.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine ftgen 0, 0, 2^10, 10, 1

instr 1

asig poscil3 .5, 880, giSine
;write a raw file: 32 bits with header
  fout "fout_880.wav", 15, asig
  outs asig, asig

endin

instr 2

klfo lfo 1, 2, 0
asig poscil3 .5*klfo, 220, giSine
;write an aiff file: 32 bits with header
  fout "fout_aif.aiff", 25, asig
;   fout "fout_all3.wav", 14, asig
  outs asig, asig

endin

instr 99 ;read the stereo csound output buffer
```

```

allL, allR monitor
;write the output of csound to an audio file
;to a wav file: 16 bits with header
      fout "fout_all.wav", 14, allL, allR

endin
</CsInstruments>
<CsScore>

i 1 0 2
i 2 0 3
i 99 0 3
e
</CsScore>
</CsoundSynthesizer>

```

Here is another example of *fout*, using it to save the contents of a table to an audio file. It uses the file *fout\_ftable.csd* [examples/fout\_ftable.csd] and *beats.wav* [examples/beats.wav].

### Example 342. Example of the *fout* opcode to save the contents of an f-table.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out      Audio in      No messages
-odac           -iadc          -d           ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o fout_ftable.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
; By: Jonathan Murphy 2007

gilen = 131072
gicps = sr/gilen
gitab ftgen 1, 0, gilen, 10, 1

      instr 1

/***** write file to table *****/

ain disk2 "beats.wav", 1, 0, 1
aphs phasor gicps
andx = aphis * gilen
tablew ain, andx, gitab

/***** write table to file *****/

aosc table aphis, gitab, 1
out aosc
fout "beats_copy.wav", 6, aosc

      endin

</CsInstruments>
<CsScore>
i1 0 2
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*fiopen, fouti, foutir, foutk, monitor*

## Credits

Author: Gabriel Maldonado

Italy

1999

Author: John ffitth

NUIM, 2013

New in Csound version 3.56

October 2002. Added a note from Richard Dobson.

Array version new in 6.01

# fouti

`fouti` — Outputs i-rate signals of an arbitrary number of channels to a specified file.

## Description

`fouti` output  $N$  i-rate signals to a specified file of  $N$  channels.

## Syntax

```
fouti ihandle, iformat, iflag, iout1 [, iout2, iout3, ..., ioutN]
```

## Initialization

*ihandle* -- a number which specifies this file.

*iformat* -- a flag to choose output file format:

- 0 - floating point in text format
- 1 - 32-bit floating point in binary format

*iflag* -- choose the mode of writing to the ASCII file (valid only in ASCII mode; in binary mode *iflag* has no meaning, but it must be present anyway). *iflag* can be a value chosen among the following:

- 0 - line of text without instrument prefix
- 1 - line of text with instrument prefix (see below)
- 2 - reset the time of instrument prefixes to zero (to be used only in some particular cases. See below)

*iout*, ..., *ioutN* -- values to be written to the file

## Performance

`fouti` and `foutir` write i-rate values to a file. The main use of these opcodes is to generate a score file during a realtime session. For this purpose, the user should set *iformat* to 0 (text file output) and *iflag* to 1, which enable the output of a prefix consisting of the strings *inum*, *actiontime*, and *duration*, before the values of *iout1*...*ioutN* arguments. The arguments in the prefix refer to instrument number, action time and duration of current note.

Notice that `fout` and `foutk` can use either a string containing a file pathname, or a handle-number generated by `fiopen`. Whereas, with `fouti` and `foutir`, the target file can be only specified by means of a handle-number.

## Examples

Here is an example of the `fouti` opcode. It uses the file `fouti.csd` [examples/fouti.csd].

### Example 343. Example of the `fouti` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o fouti.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gihand fiopen "test.txt", 0

instr 1

ires random 0, 10
fouti gihand, 0, 1, ires
ficlose gihand

endin
</CsInstruments>
<CsScore>

i 1 0 1

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*fiopen, fout, foutir, foutk*

## Credits

Author: Gabriel Maldonado  
Italy  
1999

New in Csound version 3.56

# foutir

*foutir* — Outputs i-rate signals from an arbitrary number of channels to a specified file.

## Description

*foutir* output *N* i-rate signals to a specified file of *N* channels.

## Syntax

```
foutir ihandle, iformat, iflag, iout1 [, iout2, iout3,...,ioutN]
```

## Initialization

*ihandle* -- a number which specifies this file.

*iformat* -- a flag to choose output file format:

- 0 - floating point in text format
- 1 - 32-bit floating point in binary format

*iflag* -- choose the mode of writing to the ASCII file (valid only in ASCII mode; in binary mode *iflag* has no meaning, but it must be present anyway). *iflag* can be a value chosen among the following:

- 0 - line of text without instrument prefix
- 1 - line of text with instrument prefix (see below)
- 2 - reset the time of instrument prefixes to zero (to be used only in some particular cases. See below)

*iout*,..., *ioutN* -- values to be written to the file

## Performance

*fouti* and *foutir* write i-rate values to a file. The main use of these opcodes is to generate a score file during a realtime session. For this purpose, the user should set *iformat* to 0 (text file output) and *iflag* to 1, which enable the output of a prefix consisting of the strings *inum*, *actiontime*, and *duration*, before the values of *iout1*...*ioutN* arguments. The arguments in the prefix refer to instrument number, action time and duration of current note.

The difference between *fouti* and *foutir* is that, in the case of *fouti*, when *iflag* is set to 1, the duration of the first opcode is undefined (so it is replaced by a dot). Whereas, *foutir* is defined at the end of note, so the corresponding text line is written only at the end of the current note (in order to recognize its duration). The corresponding file is linked by the *ihandle* value generated by the *fiopen* opcode. So *fouti* and *foutir* can be used to generate a Csound score while playing a realtime session.

Notice that *fout* and *foutk* can use either a string containing a file pathname, or a handle-number generated by *fiopen*. Whereas, with *fouti* and *foutir*, the target file can be only specified by means of a handle-number.

## Examples

Here is an example of the *foutir* opcode. It uses the file *foutir.csd* [examples/foutir.csd] and creates a list "foutir.sco". It can be used as a score file.

### Example 344. Example of the foutir opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0 ;;realtime audio out and midi in
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o foutir.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gihand fiopen "foutir.sco", 0

instr 1 ; play virtual midi keyboard

inot notnum ;just for printing on screen
icps cpsmidi
iamp ampmidi 1

      foutir gihand, 0, 1, icps, iamp
      prints "WRITING:\n"
      prints "note = %f,velocity = %f\n", icps, iamp ;prints them
      ficlose gihand
asig pluck iamp, icps, 1000, 0, 1
      outs asig, asig

endin
</CsInstruments>
<CsScore>

f 0 10

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*fiopen, fout, fouti, foutk*

## Credits

Author: Gabriel Maldonado  
Italy  
1999

New in Csound version 3.56

# foutk

**foutk** — Outputs k-rate signals of an arbitrary number of channels to a specified file, in raw (headerless) format.

## Description

*foutk* outputs *N* k-rate signals to a specified file of *N* channels.

## Syntax

```
foutk ifilename, iformat, kout1 [, kout2, kout3, ..., koutN]
```

## Initialization

*ifilename* -- the output file's name (in double-quotes).

*iformat* -- a flag to choose output file format (note: Csound versions older than 5.0 may only support formats 0 and 1):

- 0 - 32-bit floating point samples without header (binary PCM multichannel file)
- 1 - 16-bit integers without header (binary PCM multichannel file)
- 2 - 16-bit integers without header (binary PCM multichannel file)
- 3 - u-law samples without header
- 4 - 16-bit integers without header
- 5 - 32-bit integers without header
- 6 - 32-bit floats without header
- 7 - 8-bit unsigned integers without header
- 8 - 24-bit integers without header
- 9 - 64-bit floats without header

## Performance

*kout1, ..., koutN* -- control-rate signals to be written to the file. The expected range of the signals is determined by the selected sample format.

*foutk* operates in the same way as *fout*, but with k-rate signals. *iformat* can be set only in the range 0 to 9, or 0 to 1 with an old version of Csound.

Notice that *fout* and *foutk* can use either a string containing a file pathname, or a handle-number generated by *fiopen*. Whereas, with *fouti* and *foutir*, the target file can be only specified by means of a handle-number.

## See Also

*fiopen, fout, fouti, foutir*

## Credits

Author: Gabriel Maldonado  
Italy  
1999

New in Csound version 3.56

# fprintks

fprintks — Similar to printks but prints to a file.

## Description

Similar to *printks* but prints to a file.

## Syntax

```
fprintks "filename", "string", [, kval1] [, kval2] [...]
```

## Initialization

*"filename"* -- name of the output file.

*"string"* -- the text string to be printed. Can be up to 8192 characters and must be in double quotes.

## Performance

*kval1*, *kval2*, ... (optional) -- The k-rate values to be printed. These are specified in *"string"* with the standard C value specifier (%f, %d, etc.) in the order given.

*fprintks* is similar to the *printks* opcode except it outputs to a file and doesn't have a *itime* parameter. For more information about output formatting, please look at *printks's documentation*.

## Examples

Here is an example of the fprintks opcode. It uses the file *fprintks.csd* [examples/fprintks.csd].

### Example 345. Example of the fprintks opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o fprintks.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

/* Written by Matt Ingalls, edited by Kevin Conder. */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - a score generator example.
instr 1
; K-rate stuff.
kstart init 0
kdur linrand 10
```

```

kpitch linrand 8

; Printing to to a file called "my.sco".
fprintks "my.sco", "i1\\t%2.2f\\t%2.2f\\t%2.2f\\n", kstart, kdur, 4+kpitch

knext linrand 1
kstart = kstart + knext
endin

</CsInstruments>
<CsScore>

/* Written by Matt Ingalls, edited by Kevin Conder. */
; Play Instrument #1.
i 1 0 0.001

</CsScore>
</CsoundSynthesizer>

```

This example will generate a file called “my.sco”. It should contain lines like this:

```

i1      0.00    3.94    10.26
i1      0.20    3.35    6.22
i1      0.67    3.65    11.33
i1      1.31    1.42    4.13

```

Here is an example of the fprintks opcode, which converts a standard MIDI file to a csound score. It uses the file *fprintks-2.csd* [examples/fprintks-2.csd].

### Example 346. Example of the fprintks opcode to convert a MIDI file to a csound score.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
; -odac      -iadc     -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
-n -Fmidichn_advanced.mid
;Don't write audio ouput to disk and use the file midichn_advanced.mid as MIDI input
</CsOptions>
<CsInstruments>

sr      = 48000
ksmps   = 16
nchnls  = 2

;Example by Jonathan Murphy 2007

; assign all midi events to instr 1000
massign 0, 1000
pgmassign 0, 1000

instr 1000

ktim timeinsts

kst, kch, kd1, kd2 midiin
if (kst != 0) then
; p4 = MIDI event type  p5 = channel  p6= data1  p7= data2
fprintks "MIDI2cs.sco", "i1\\t%f\\t%f\\t%d\\t%d\\t%d\\t%d\\n", ktim, 1/kr, kst, kch, kd1, kd2

```

```

endif

    endin

</CsInstruments>
<CsScore>
i1000 0 10000
e
</CsScore>
</CsoundSynthesizer>

```

This example will generate a file called “MIDI2cs.sco” containing i-events according to the MIDI file

Here is an advanced example of the `fprintks` opcode, which generates scores for Csound. It uses the file `scogen-2.csd` [examples/scogen.csd].

### Example 347. Example of the `fprintks` opcode to create a Csound score file generator using Csound.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
; -odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
-n
;Don't write audio ouput to disk
</CsOptions>
<CsInstruments>
;=====
;      scogen.csd      by: Matt Ingalls
;
;  a "port" of sorts
;  of the old "mills" score generator (scogen)
;
;  this instrument creates a schottstaedt.sco file
;  to be used with the schottstaedt.orc file
;
;  as long as you dont save schottstaedt.orc as a .csd
;  file, you should be able to keep it open in MacCsound
;  and render each newly generated .sco file.
;
;=====

gScoName = "/Users/matt/Desktop/schottstaedt.sco"      ; the name of the file to be generated

    sr      =      100      ; this defines our temporal resolution,
                        ; an sr of 100 means we will generate p2 and p3 values
                        ; to the nearest 1/100th of a second

    ksmps =      1      ; set kr=sr so we can do everything at k-rate

; some print opcodes
opcode PrintInteger, 0, k
    kval      xin
    fprintks      gScoName, "%d", kval
endop

opcode PrintFloat, 0, k
    kval      xin
    fprintks      gScoName, "%f", kval
endop

```

```

opcode PrintTab, 0, 0
    fprintks    gScoName, "%n"
endop

opcode PrintReturn, 0, 0
    fprintks    gScoName, "%r"
endop

; recursively calling opcode to handle all the optional parameters
opcode ProcessAdditionalPfields, 0, ikio
    iPtable, kndx, iNumPfields, iPfield xin

    ; additional pfields start at 5, we use a default 0 to identify the first call
    iPfield = (iPfield == 0 ? 5 : iPfield)

    if (iPfield > iNumPfields) goto endloop
        ; find our tables
        iMinTable table    2*iPfield-1, iPtable
        iMaxTable table    2*iPfield, iPtable

        ; get values from our tables
        kMin tablei    kndx, iMinTable
        kMax tablei    kndx, iMaxTable

        ; find a random value in the range and write it to the score
        fprintks gScoName, "%t%f", kMin + rnd(kMax-kMin)

        ; recursively call for any additional pfields.
        ProcessAdditionalPfields iPtable, kndx, iNumPfields, iPfield + 1
    endloop:
endop

/* =====
Generate a gesture of i-statements

p2 = start of the gesture
p3 = duration of the gesture
p4 = number of a function that contains a list of all
function table numbers used to define the
pfield random distribution
p5 = scale generated p4 values according to density (0=off, 1=on) [todo]
p6 = let durations overlap gesture duration (0=off, 1=on) [todo]
p7 = seed for random number generator seed [todo]
=====
*/
instr Gesture

; initialize
iResolution = 1/sr

kNextStart init p2
kCurrentTime init p2

iNumPfields table    0, p4
iInstrMinTable table  1, p4
iInstrMaxTable table  2, p4
iDensityMinTable table  3, p4
iDensityMaxTable table  4, p4
iDurMinTable table    5, p4
iDurMaxTable table    6, p4
iAmpMinTable table    7, p4
iAmpMaxTable table    8, p4

```

```

; check to make sure there is enough data
print iNumPfields
if iNumPfields < 4 then
    prints "%dError: At least 4 p-fields (8 functions) need to be specified.%n", iNumPfields
    turnoff
endif

; initial comment
fprints   gScoName, "%!Generated Gesture from %f to %f seconds%n %!t%twith a p-max of %d%n%n", p2

; k-rate stuff
if (kCurrentTime >= kNextStart) then ; write a new note!

    kndx = (kCurrentTime-p2)/p3

    ; get the required pfield ranges
    kInstMin tablei   kndx, iInstrMinTable
    kInstMax tablei   kndx, iInstrMaxTable
    kDensMin tablei   kndx, iDensityMinTable
    kDensMax tablei   kndx, iDensityMaxTable
    kDurMin tablei    kndx, iDurMinTable
    kDurMax tablei    kndx, iDurMaxTable
    kAmpMin tablei    kndx, iAmpMinTable
    kAmpMax tablei    kndx, iAmpMaxTable

    ; find random values for all our required parameters and print the i-statement
    fprintks gScoName, "i%d%t%f%t%f%t%f", kInstMin + rnd(kInstMax-kInstMin), kNextStart, kDurMin +

    ; now any additional pfields
    ProcessAdditionalPfields p4, kndx, iNumPfields

    PrintReturn

    ; calculate next starttime
    kDensity = kDensMin + rnd(kDensMax-kDensMin)
    if (kDensity < iResolution) then
        kDensity = iResolution
    endif
    kNextStart = kNextStart + kDensity
endif

kCurrentTime = kCurrentTime + iResolution
endin

</CsInstruments>
<CsScore>
/*
=====
scogen.sco

    this csound module generates a score file
    you specify a gesture of notes by giving
    the "gesture" instrument a number to a
    (negative) gen2 table.

    this table stores numbers to pairs of functions.
    each function-pair represents a range (min-max)
    of randomness for every pfield for the notes to
    be generated.
=====
*/

; common tables for pfield ranges
f100  0  2  -7  0 2 0  ; static 0
f101  0  2  -7  1 2 1  ; static 1

```

```

f102  0  2  -7  0 2 1 ; ramp 0->1
f103  0  2  -7  1 2 0 ; ramp 1->0
f105  0  2  -7  10 2 10 ; static 10
f106  0  2  -7  .1 2 .1 ; static .1

; specific pfield ranges
f10  0  2  -7  .8 2 .01 ; density
f11  0  2  -7  8 2 4 ; pitchmin
f12  0  2  -7  8 2 12 ; pitchmax

;=== table containing the function numbers used for all the p-field distributions
;
; p1 - table number
; p2 - time table is instantiated
; p3 - size of table (must be >= p5!)
; p4 - gen# (should be = -2)
; p5 - number of pfields of each note to be generated
; p6 - table number of the function representing the minimum possible note number (p1) of a generation
; p7 - table number of the function representing the maximum possible note number (p1) of a generation
; p8 - table number of the function representing the minimum possible noteon-to-noteon time (p2)
; p9 - table number of the function representing the maximum possible noteon-to-noteon time (p2)
; p10 - table number of the function representing the minimum possible duration (p3) of a generation
; p11 - table number of the function representing the maximum possible duration (p3) of a generation
; p12 - table number of the function representing the maximum possible amplitude (p4) of a generation
; p13 - table number of the function representing the maximum possible amplitude (p5) of a generation
; p14,p16.. - table number of the function representing the minimum possible value for additional pfields
; p15,p17.. - table number of the function representing the maximum possible value for additional pfields

;      siz  2  #pds p1min  p1max p2min  p2max p3min  p3max p4min  p4max p5min  p5max p6min  p6max
f1  0  32  -2  6  101  101  10  10 101  105  100  106  11  12  100  101

;gesture definitions
;      start  dur  pTble  scale  overlap seed
i"Gesture"  0  60  1 ;todo-->0  0  123
</CsScore>

</CsoundSynthesizer>

```

This example will generate a file called “schottstaedt.sco” which can be used as a score together with *schottstaedt.orc* [examples/schottstaedt.orc]

## See Also

*printks*

## Credits

Author: Matt Ingalls  
 January 2003

# fprints

fprints — Similar to prints but prints to a file.

## Description

Similar to *prints* but prints to a file.

## Syntax

```
fprints "filename", "string" [, ival1] [, ival2] [...]
```

## Initialization

*"filename"* -- name of the output file.

*"string"* -- the text string to be printed. Can be up to 8192 characters and must be in double quotes.

*ival1, ival2, ...* (optional) -- The i-rate values to be printed. These are specified in *"string"* with the standard C value specifier (%f, %d, etc.) in the order given.

## Performance

*fprints* is similar to the *prints* opcode except it outputs to a file. For more information about output formatting, please look at *prints's* documentation.

## Examples

Here is an example of the *fprints* opcode. It uses the file *fprints.csd* [examples/fprints.csd]. Note that the example must be run from a directory for which the user has write access.

### Example 348. Example of the fprints opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      -iadc      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o fprints.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

/* Written by Matt Ingalls, edited by Kevin Conder. */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1
```

```
; Instrument #1 - a score generator example.
instr 1
  ; Print to the file "my.sco".
  fprints "my.sco", "%!Generated score by ma++\n\n"
endin

</CsInstruments>
<CsScore>

/* Written by Matt Ingalls, edited by Kevin Conder. */
; Play Instrument #1.
i 1 0 0.001

</CsScore>
</CsoundSynthesizer>
```

This example will generate a file called “my.sco”. It should contain a line like this:

```
!Generated score by ma++
```

## See Also

*prints*

## Credits

Author: Matt Ingalls  
January 2003

# frac

frac — Returns the fractional part of a decimal number.

## Description

Returns the fractional part of  $x$ .

## Syntax

`frac(x)` (init-rate or control-rate args; also works at audio rate in Csound5)

`frac(k/i[[]])` (k- or i-arrays )

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

## Examples

Here is an example of the frac opcode. It uses the file `frac.csd` [examples/frac.csd].

### Example 349. Example of the frac opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o frac.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  i1 = 16 / 5
  i2 = frac(i1)

  print i2
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
e
```

```
</CsScore>  
</CsoundSynthesizer>
```

Its output should include a line like this:

```
instr 1: i2 = 0.200
```

## See Also

*abs, exp, int, log, log10, i, sqrt*

## Credits

Example written by Kevin Conder.

# fractalnoise

fractalnoise — A fractal noise generator.

## Description

A fractal noise generator implemented as a white noise filtered by a cascade of 15 first-order filters.

## Syntax

```
ares fractalnoise kamp, kbeta
```

## Performance

*kamp* -- amplitude.

*kbeta* -- spectral parameter related to the fractal dimension

- 0 - white
- 1 - pink
- 2 - brown

## Examples

Here is an example of the fractalnoise opcode. It uses the file *fractalnoise.csd* [examples/fractalnoise.csd].

### Example 350. Example of the fractalnoise opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o oscil.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kbeta linseg 0, p3/4, 2, p3/4, 0, p3*0.1, 2, p3*0.15, 0
seed 20120124
aout fractalnoise 0.05, kbeta
outs aout, aout
```

```
endin
</CsInstruments>
<CsScore>
i1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

## References

1. R. Saletti. A comparison between two methods to generate  $1/(f^\gamma)$  noise. In Proc. IEEE, volume 74, pp. 1595-1596, November 1986.
2. G. Corsini and R. Saletti. A  $1/(f^\gamma)$  power spectrum noise sequence generator. IEEE Trans. on Instrumentation and Measurement, 37(4):615-619, December 1988.
3. The Sounding Object, edited by Davide Rocchesso and Federico Fontana, Edizioni di Mondo Estremo. Chapter 8 by Federico Avanzini, pp. 154-157.

## Credits

Author: Tito Latini  
January 2012

New in Csound version 5.16

# framebuffer

`framebuffer` — Read audio signals into 1 dimensional k-rate arrays and vice-versa with a specified buffer size.

## Description

*framebuffer* converts audio signals into a 1 dimensional k-rate array of a specified size. The size of the k-rate array must be larger than `ksmps`. It also converts 1 dimensional k-rate arrays back into audio signals, the k-rate arrays must be uninitialised and also be larger than `ksmps` in size. The buffer works as a circular buffer and may be used for doing frame based audio processing such as spectral analysis/synthesis or as a simple delay line.

## Syntax

```
kout[] framebuffer ain, isize
```

```
aout framebuffer kin, isize
```

## Initialization

*isize* -- The amount of samples that the buffer will contain.

## Performance

*kout[]* -- The k-rate array output from the framebuffer. *ain* -- The audio signal input to the framebuffer.

*aout* -- The audio signal output from the framebuffer. *kin* -- The k-rate array input to the framebuffer.

## Examples

Here is a simple example of the framebuffer opcode. It uses the file *framebuffer.csd* [examples/framebuffer.csd].

### Example 351. Example of the framebuffer opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
nchnls = 2
0dbfs = 1
ksmps = 128
sr = 44100

instr 1
  isize init 1024
  ioverlaps init 4
```

```
asig diskin2 "fox.wav", 1, 0, 1
kframe[] framebuffer asig, isize
kwindowedFrame[] window kframe, isize

aout olabuffer kwindowedFrame, ioverlaps
aout = aout / 2

outs aout, aout
endin

</CsInstruments>
<CsScore>
i 1 0 400
</CsScore>
</CsoundSynthesizer>
```

## See Also

*olabuffer shiftin shiftout*

## Credits

Author: Edward Costello;  
NUIM, 2015

New in version 6.06

# freeverb

freeverb — Opcode version of Jezar's Freeverb

## Description

freeverb is a stereo reverb unit based on Jezar's public domain C++ sources, composed of eight parallel comb filters on both channels, followed by four allpass units in series. The filters on the right channel are slightly detuned compared to the left channel in order to create a stereo effect.

## Syntax

```
aoutL, aoutR freeverb ainL, ainR, kRoomSize, kHFDamp[, iSRate[, iSkip]]
```

## Initialization

*iSRate* (optional, defaults to 44100): adjusts the reverb parameters for use with the specified sample rate (this will affect the length of the delay lines in samples, and, as of the latest CVS version, the high frequency attenuation). Only integer multiples of 44100 will reproduce the original character of the reverb exactly, so it may be useful to set this to 44100 or 88200 for an orchestra sample rate of 48000 or 96000 Hz, respectively. While *iSRate* is normally expected to be close to the orchestra sample rate, different settings may be useful for special effects.

*iSkip* (optional, defaults to zero): if non-zero, initialization of the opcode will be skipped, whenever possible.

## Performance

*ainL*, *ainR* -- input signals; usually both are the same, but different inputs can be used for special effect



### Note

It is recommended to process the input signal(s) with the `denorm` opcode in order to avoid denormalized numbers which could significantly increase CPU usage in some cases

*aoutL*, *aoutR* -- output signals for left and right channel

*kRoomSize* (range: 0 to 1) -- controls the length of the reverb, a higher value means longer reverb. Settings above 1 may make the opcode unstable.

*kHFDamp* (range: 0 to 1): high frequency attenuation; a value of zero means all frequencies decay at the same rate, while higher settings will result in a faster decay of the high frequency range.

## Examples

Here is an example of the *freeverb* opcode. It uses the file *freeverb.csd* [examples/freeverb.csd].

### Example 352. An example of the freeverb opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o freeverb.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
sr      = 44100
ksmps   = 32
nchnls  = 2
0dbfs   = 1

      instr 1
a1      vco2 0.75, 440, 10
kfrq    port 100, 0.008, 20000
a1      butterlp a1, kfrq
a2      linseg 0, 0.003, 1, 0.01, 0.7, 0.005, 0, 1, 0
a1      = a1 * a2
      denorm a1
aL, aR  freeverb a1, a1, 0.9, 0.35, sr, 0
      outs a1 + aL, a1 + aR
      endin

</CsInstruments>
<CsScore>
i 1 0 5
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Istvan Varga  
2005

# ftchnls

ftchnls — Returns the number of channels in a stored function table.

## Description

Returns the number of channels in a stored function table.

## Syntax

```
ftchnls(x) (init-rate args only)
```

## Performance

Returns the number of channels of a *GEN01* table, determined from the header of the original file. If the original file has no header or the table was not created by these *GEN01*, *ftchnls* returns -1.

## Examples

Here is an example of the *ftchnls* opcode. It uses the files *ftchnls.csd* [examples/ftchnls.csd], *mary.wav* [examples/mary.wav], and *kickroll.wav* [examples/kickroll.wav].

### Example 353. Example of the *ftchnls* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
;-odac      ;;realtime audio out
-iadc      ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o ftchnls.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ichnls = ftchnls(p4)
print ichnls

if (ichnls == 1) then
  asigL loscil3 .8, 4, p4
  asigR = asigL
elseif (ichnls == 2) then
  asigL, asigR loscil3 .8, 4, p4
; safety precaution if not mono or stereo
else
  asigL = 0
  asigR = 0
```

```
endif      outs asigL, asigR

endin
</CsInstruments>
<CsScore>
f 1 0 0 1 "mary.wav" 0 0 0
f 2 0 0 1 "kickroll.wav" 0 0 0

i 1 0 3 1 ;mono file
i 1 + 2 2 ;stereo file
e
</CsScore>
</CsoundSynthesizer>
```

The audio file “mary.wav” is monophonic, the audio file “kickroll.wav” is stereophonic, so its output should include lines like these:

```
instr 1:  ichnls = 1.000
instr 1:  ichnls = 2.000
```

## See Also

*flen, ftlptim, ftsr, nsamp*

## Credits

Author: Chris McCormick  
Perth, Australia  
December 2001

# ftconv

ftconv — Low latency multichannel convolution, using a function table as impulse response source.

## Description

Low latency multichannel convolution, using a function table as impulse response source. The algorithm is to split the impulse response to partitions of length determined by the *iplen* parameter, and delay and mix partitions so that the original, full length impulse response is reconstructed without gaps. The output delay (latency) is *iplen* samples, and does not depend on the control rate, unlike in the case of other convolve opcodes.

## Syntax

```
a1[, a2[, a3[, ... a8]]] ftconv ain, ift, iplen[, iskip samples \  
[, iirlen[, iskipinit]]]
```

## Initialization

*ift* -- source ftable number. The table is expected to contain interleaved multichannel audio data, with the number of channels equal to the number of output variables (*a1*, *a2*, etc.). An interleaved table can be created from a set of mono tables with *GEN52*.

*iplen* -- length of impulse response partitions, in sample frames; must be an integer power of two. Lower settings allow for shorter output delay, but will increase CPU usage.

*iskipsamples* (optional, defaults to zero) -- number of sample frames to skip at the beginning of the table. Useful for reverb responses that have some amount of initial delay. If this delay is not less than *iplen* samples, then setting *iskipsamples* to the same value as *iplen* will eliminate any additional latency by *ftconv*.

*iirlen* (optional) -- total length of impulse response, in sample frames. The default is to use all table data (not including the guard point).

*iskipinit* (optional, defaults to zero) -- if set to any non-zero value, skip initialization whenever possible without causing an error.

## Performance

*ain* -- input signal.

*a1* ... *a8* -- output signal(s).

## Example

Here is an example of the *ftconv* opcode. It uses the file *ftconv.csd* [examples/ftconv.csd].

### Example 354. Example of the *ftconv* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o ftconv.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
sr          = 48000
ksmps      = 32
nchnls    = 2
0dbfs     = 1

garvb      init 0
gaW        init 0
gaX        init 0
gaY        init 0

itmp      ftgen  1, 0, 64, -2, 2, 40, -1, -1, -1, 123,      \
            1, 13.000, 0.05, 0.85, 20000.0, 0.0, 0.50, 2,  \
            1, 2.000, 0.05, 0.85, 20000.0, 0.0, 0.25, 2,  \
            1, 16.000, 0.05, 0.85, 20000.0, 0.0, 0.35, 2,  \
            1, 9.000, 0.05, 0.85, 20000.0, 0.0, 0.35, 2,  \
            1, 12.000, 0.05, 0.85, 20000.0, 0.0, 0.35, 2,  \
            1, 8.000, 0.05, 0.85, 20000.0, 0.0, 0.35, 2

itmp      ftgen  2, 0, 262144, -2, 0
spat3dt  2, -0.2, 1, 0, 1, 1, 2, 0.005

itmp      ftgen  3, 0, 262144, -52, 3, 2, 0, 4, 2, 1, 4, 2, 2, 4

instr 1

a1        vco2 1, 440, 10
kfrq      port 100, 0.008, 20000
a1        butterlp a1, kfrq
a2        linseg 0, 0.003, 1, 0.01, 0.7, 0.005, 0, 1, 0
a1        = a1 * a2 * 2
denorm    a1
vincr     garvb, a1
aw, ax, ay, az spat3di a1, p4, p5, p6, 1, 1, 2
vincr     gaW, aw
vincr     gaX, ax
vincr     gaY, ay

endin

instr 2

denorm    garvb
; skip as many samples as possible without truncating the IR
arW, arX, arY ftconv garvb, 3, 2048, 2048, (65536 - 2048)
aW        = gaW + arW
aX        = gaX + arX
aY        = gaY + arY
garvb     = 0
gaW       = 0
gaX       = 0
gaY       = 0

aWre, aWim hilbert aW
aXre, aXim hilbert aX
aYre, aYim hilbert aY
aWXr     = 0.0928*aXre + 0.4699*aWre

```

```
aWxiYr = 0.2550*aXim - 0.1710*aWim + 0.3277*aYre  
aL      = aWXr + aWxiYr  
aR      = aWXr - aWxiYr
```

```
outs aL, aR
```

```
endin
```

```
</CsInstruments>
```

```
<CsScore>
```

```
i 1 0 0.5 0.0 2.0 -0.8
```

```
i 1 1 0.5 1.4 1.4 -0.6
```

```
i 1 2 0.5 2.0 0.0 -0.4
```

```
i 1 3 0.5 1.4 -1.4 -0.2
```

```
i 1 4 0.5 0.0 -2.0 0.0
```

```
i 1 5 0.5 -1.4 -1.4 0.2
```

```
i 1 6 0.5 -2.0 0.0 0.4
```

```
i 1 7 0.5 -1.4 1.4 0.6
```

```
i 1 8 0.5 0.0 2.0 0.8
```

```
i 2 0 10
```

```
e
```

```
</CsScore>
```

```
</CsoundSynthesizer>
```

## See also

*pconvolve*, *convolve*, *dconv*.

## Credits

Author: Istvan Varga

2005

# ftcps

ftcps — Returns the base frequency of a stored function table in Hz.

## Description

Returns the base frequency of a stored function table in Hz..

## Syntax

```
ftcps(x) (init-rate args only)
```

## Performance

Returns the base frequency of stored function table, number *x*. *ftcps* is designed for tables storing audio waveforms read from files (see *GENOI*).

*ftcps* returns -1 in case of an error (no base frequency set in table or non-existent table).

## Examples

Here is an example of the ftcps opcode.

### Example 355. Example of the ftcps opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o ftlen.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Print out the base frequency of Table #1.
; if it has been set in the original file.
icps = ftcps(1)
print icps
endin

</CsInstruments>
<CsScore>
```

```
; Table #1: Use an audio file, Csound will determine its base frequency, if set.
f 1 0 0 1 "sample.wav" 0 0 0

; Play Instrument #1 for 1 second.
i 1 0 1
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*ftchnls, flptim, ftsr, nsamp*

## Credits

Author: Victor Lazzarini  
2010

Example written by Victor Lazzarini

# ftfree

ftfree — Deletes function table.

## Description

Deletes function table.

## Syntax

```
ftfree ifno, iwhen
```

## Initialization

*ifno* -- the number of the table to be deleted

*iwhen* -- if zero the table is deleted at init time; otherwise the table number is registered for being deleted at note deactivation.

## Examples

Here is an example of the `ftfree` opcode. It uses the file `ftfree.csd` [examples/ftfree.csd].

### Example 356. Example of the `ftfree` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o ftfree.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gitempTable ftgen 0, 0, 65537, 10, 1

instr 1

aout oscili .5, 440, gitempTable
outs aout, aout

;free temp table at deinit time
ftfree gitempTable, 1
print gitempTable

endin
</CsInstruments>
```

```
<CsScore>
f 0 5

i 1 0 .1
i 1 3 1

e
</CsScore>
</CsoundSynthesizer>

instr 1: gitempTable = 101.000
B 0.000 .. 3.000 T 3.000 TT 3.000 M: 0.50000 0.50000
INIT ERROR in instr 1: Invalid ftable no. 101.000000
instr 1: gitempTable = 101.000
Error deleting ftable 101
  B 3.000 - note deleted. i1 had 1 init errors
B 3.000 .. 5.000 T 5.000 TT 5.000 M: 0.00000 0.00000
```

## Credits

Authors: Steven Yi, Istvan Varga  
2005

# ftgen

ftgen — Generate a score function table from within the orchestra.

## Description

Generate a score function table from within the orchestra.

## Syntax

```
gir ftgen ifn, itime, isize, igen, iarga [, iargb ] [...]  
gir ftgen ifn, itime, isize, igen, iarray
```

## Initialization

*gir* -- either a requested or automatically assigned table number above 100.

*ifn* -- requested table number If *ifn* is zero, the number is assigned automatically and the value placed in *gir*. Any other value is used as the table number

*itime* -- is ignored, but otherwise corresponds to p2 in the score *f statement*.

*isize* -- table size. Corresponds to p3 of the score *f statement*.

*igen* -- function table *GEN* routine. Corresponds to p4 of the score *f statement*.

*iarga*, *iargb*, ... -- function table arguments. Correspond to p5 through *pn* of the score *f statement*.

*iarray* -- ane dimensionl array holding the function table arguments. Correspond to p5 through *pn* of the score *f statement*.

## Performance

This is equivalent to table generation in the score with the *f statement*.



### Note

Csound was originally designed to support tables with power of two sizes only. Though this has changed in recent versions (you can use any size by using a negative number), many opcodes will not accept them.



### Warning

Although Csound will not protest if ftgen is used inside instr-endin statements, this is not the intended or supported use, and must be handled with care as it has global effects. (In particular, a different size usually leads to relocation of the table, which may cause a crash or otherwise erratic behaviour.

## Examples

Here is an example of the ftgen opcode. It uses the file *ftgen.csd* [examples/ftgen.csd].

**Example 357. Example of the ftgen opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o ftgen.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gisine  ftgen 1, 0, 16384, 10, 1 ;sine wave
gisquare ftgen 2, 0, 16384, 10, 1, 0, .33, 0, .2, 0, .14, 0, .11, 0, .09 ;odd harmonics
gisaw   ftgen 3, 0, 16384, 10, 0, .2, 0, .4, 0, .6, 0, .8, 0, 1, 0, .8, 0, .6, 0, .4, 0, .2 ;even harmo

instr 1

ifn = p4
asig poscil .6, 200, ifn
    outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 2 1 ;sine wave
i 1 3 2 2 ;odd harmonics
i 1 6 2 3 ;even harmonics
e
</CsScore>
</CsoundSynthesizer>

```

Here is another example of the ftgen opcode. It uses the file *ftgen-2.csd* [examples/ftgen-2.csd].

**Example 358. Example of the ftgen opcode.**

This example queries a file for its length to create an f-table of the appropriate size.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in
-odac      -iadc     ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o ftgen-2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr      = 48000
ksmps   = 16
nchnls  = 2

;Example by Jonathan Murphy 2007

```

```
Odbfs      = 1

instr 1

Sfile     = "beats.wav"

ilen      filelen Sfile ; Find length
isr       filesr  Sfile ; Find sample rate

isamps    = ilen * isr ; Total number of samples
isize     init    1

loop:
  isize   = isize * 2
  ; Loop until isize is greater than number of samples
  if (isize < isamps) igoto loop

  itab    ftgen    0, 0, isize, 1, Sfile, 0, 0, 0
  print   isize
  print   isamps

  turnoff
  endin

</CsInstruments>
<CsScore>
i1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

## See also

*GEN routine overview, ftgentmp*

## Credits

Author: Barry L. Vercoe  
M.I.T., Cambridge, Mass  
1997

Added warning April 2002 by Rasmus Ekman

# ftgenonce

ftgenonce — Generate a function table from within an instrument definition, without duplication of data.

## Description

Enables the creation of function tables entirely inside instrument definitions, without any duplication of data.

The ftgenonce opcode is designed to simplify writing instrument definitions that can be re-used in different orchestras simply by #including them and plugging them into some output instrument. There is no need to define function tables either in the score, or in the orchestra header.

The ftgenonce opcode is similar to ftgentmp, and has identical arguments. However, function tables are neither duplicated nor deleted. Instead, all of the arguments to the opcode are concatenated to form the key to a lookup table that points to the function table number. Thus, every request to ftgenonce with the same arguments receives the same instance of the function table data. Every change in the value of any ftgenonce argument causes the creation of a new function table.

## Syntax

```
ifno ftgenonce ip1, ip2dummy, isize, igen, iarga, iargb, ...
```

## Initialization

*ifno* -- an automatically assigned table number.

*ip1* -- the number of the table to be generated or 0 if the number is to be assigned.

*ip2dummy* -- ignored.

*isize* -- table size. Corresponds to p3 of the score *f statement*.

*igen* -- function table *GEN* routine. Corresponds to p4 of the score *f statement*.

*iarga, iargb, ...* -- function table arguments. Correspond to p5 through *pn* of the score *f statement*.



### Note

Csound was originally designed to support tables with power of two sizes only. Though this has changed in recent versions (you can use any size by using a negative number), many opcodes will not accept them.

## Examples

Here is an example of the ftgenonce opcode. It uses the file *ftgenonce.csd* [examples/ftgenonce.csd].

### Example 359. Example of the ftgenonce opcode.

```
<CsoundSynthesizer>
<CsOptions>
```

```
; Select audio/midi flags here according to platform
-odac      ;;;realtime audio out
;-iadc     ;;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o ftgenonce.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
; Use ftgenonce instead of ftgen, ftgentmp, or f statement
iHz = p4
isine ftgenonce 0, 0, 1024, 10, 1
aoscili pluck .7, iHz, 100, isine, 1
aadsr adsr 0.015, 0.07, 0.6, 0.3
asig = aoscili * aadsr
      outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 1 220
i 1 2 1 261
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Authors: Michael Gogins  
2009

# ftgentmp

`ftgentmp` — Generate a score function table from within the orchestra, which is deleted at the end of the note.

## Description

Generate a score function table from within the orchestra, which is optionally deleted at the end of the note.

## Syntax

```
ifno ftgentmp ip1, ip2dummy, isize, igen, iarga, iargb, ...
```

## Initialization

*ifno* -- either a requested or automatically assigned table number above 100.

*ip1* -- the number of the table to be generated or 0 if the number is to be assigned, in which case the table is deleted at the end of the note activation.

*ip2dummy* -- ignored.

*isize* -- table size. Corresponds to p3 of the score *f statement*.

*igen* -- function table *GEN* routine. Corresponds to p4 of the score *f statement*.

*iarga, iargb, ...* -- function table arguments. Correspond to p5 through *pn* of the score *f statement*.



### Note

Csound was originally designed to support tables with power of two sizes only. Though this has changed in recent versions (you can use any size by using a negative number), many opcodes will not accept them.

## Examples

Here is an example of the `ftgentmp` opcode. It uses the file `ftgentmp.csd` [examples/ftgentmp.csd].

### Example 360. Example of the `ftgentmp` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 128
nchnls = 2
0dbfs = 1
```

```

instr 1
ifno ftgentmp 0, 0, 512, 10, 1
print ifno
endin

instr 2
print ftlen(p4)
endin

</CsInstruments>
<CsScore>
i 1 0 10
i 2 2 1 101
i 1 5 10
i 2 7 1 102
i 2 12 1 101
i 2 17 1 102
e
</CsScore>
</CsoundSynthesizer>

```

The output of this csd shows that the tables are destroyed after the instrument instances that created them terminates, causing an init error if the tables are queried.

```

SECTION 1:
new alloc for instr 1:
ftable 101:
instr 1: ifno = 101.000
B 0.000 .. 2.000 T 2.000 TT 2.000 M: 0.00000 0.00000
new alloc for instr 2:
instr 2: #i0 = 512.000
B 2.000 .. 5.000 T 5.001 TT 5.001 M: 0.00000 0.00000
new alloc for instr 1:
ftable 102:
instr 1: ifno = 102.000
B 5.000 .. 7.000 T 7.001 TT 7.001 M: 0.00000 0.00000
instr 2: #i0 = 512.000
B 7.000 .. 12.000 T 11.999 TT 11.999 M: 0.00000 0.00000
INIT ERROR in instr 2: Invalid ftable no. 101.000000
#i0 ftlen.i p4
instr 2: #i0 = -1.000
      B 12.000 - note deleted. i2 had 1 init errors
B 12.000 .. 17.000 T 17.000 TT 17.000 M: 0.00000 0.00000
INIT ERROR in instr 2: Invalid ftable no. 102.000000
#i0 ftlen.i p4
instr 2: #i0 = -1.000
      B 17.000 - note deleted. i2 had 1 init errors

```

## Credits

Authors: Istvan Varga  
2005

# ftlen

ftlen — Returns the size of a stored function table.

## Description

Returns the size of a stored function table.

## Syntax

```
ftlen(x) (init-rate args only)
```

## Performance

Returns the size (number of points, excluding guard point) of stored function table, number *x*. While most units referencing a stored table will automatically take its size into account (so tables can be of arbitrary length), this function reports the actual size if that is needed. Note that *ftlen* will always return a power-of-2 value, i.e. the function table guard point (see *f Statement*) is not included. As of Csound version 3.53, *ftlen* works with deferred function tables (see *GENO1*).

*ftlen* differs from *nsamp* in that *nsamp* gives the number of sample frames loaded, while *ftlen* gives the total number of samples without the guard point. For example, with a stereo sound file of 10000 samples, *ftlen()* would return 19999 (i.e. a total of 20000 mono samples, not including a guard point), but *nsamp()* returns 10000.

## Examples

Here is an example of the *ftlen* opcode. It uses the file *ftlen.csd* [examples/ftlen.csd], *fox.wav* [examples/fox.wav] and *beats.wav* [examples/beats.wav].

### Example 361. Example of the *ftlen* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o ftlen.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs  =1

instr 1

ift = ftlen(p4)
print ift
```

```
  aout loscil3 .8, 4, p4
      outs aout, aout

  endin
</CsInstruments>
<CsScore>
f 1 0 0 1 "fox.wav" 0 0 0 ;Csound computes tablesize
f 2 0 0 1 "beats.wav" 0 0 0 ;Csound computes tablesize

i 1 0 3 1 ;"fox.wav"
i 1 3 3 2 ;"beats.wav"

e
</CsScore>
</CsoundSynthesizer>
```

The audio file “fox.wav” is 121569 samples long, “beats.wav” has a length of 88200 samples. The `ftlen` opcode reports them as 121568 and 88199 samples, because it reserves 1 point for the guard point. Its output should include lines like these:

```
instr 1:  ift = 121568.000
instr 1:  ift = 88199.000
```

## See Also

*ftchnls, filptim, ftsr, nsamp*

## Credits

Author: Barry L. Vercoe  
MIT  
Cambridge, Massachusetts  
1997

# ftload

ftload — Load a set of previously-allocated tables from a file.

## Description

Load a set of previously-allocated tables from a file.

## Syntax

```
ftload "filename", iflag, ifn1 [, ifn2] [...]
```

## Initialization

*"filename"* -- A quoted string containing the name of the file to load.

*iflag* -- Type of the file to load/save. (0 = binary file, Non-zero = text file)

*ifn1, ifn2, ...* -- Numbers of tables to load.

## Performance

*ftload* loads a list of tables from a file. (The tables have to be already allocated though.) The file's format can be binary or text.



### Warning

The file's format is not compatible with a WAV-file and is not endian-safe.

## Examples

See the example for *ftsave*.

## See Also

*ftloadk, ftsavek, ftsave*

## Credits

Author: Gabriel Maldonado

New in version 4.21

# ftloadk

ftloadk — Load a set of previously-allocated tables from a file.

## Description

Load a set of previously-allocated tables from a file.

## Syntax

```
ftloadk "filename", ktrig, iflag, ifn1 [, ifn2] [...]
```

## Initialization

*"filename"* -- A quoted string containing the name of the file to load.

*iflag* -- Type of the file to load/save. (0 = binary file, Non-zero = text file)

*ifn1, ifn2, ...* -- Numbers of tables to load.

## Performance

*ktrig* -- The trigger signal. Load the file each time it is non-zero.

*ftloadk* loads a list of tables from a file. (The tables have to be already allocated though.) The file's format can be binary or text. Unlike *ftload*, the loading operation can be repeated numerous times within the same note by using a trigger signal.



### Warning

The file's format is not compatible with a WAV-file and is not endian-safe.

## See Also

*ftload, ftsavek, ftsave*

## Credits

Author: Gabriel Maldonado

New in version 4.21

# ftlptim

ftlptim — Returns the loop segment start-time of a stored function table number.

## Description

Returns the loop segment start-time of a stored function table number.

## Syntax

```
ftlptim(x) (init-rate args only)
```

## Performance

Returns the loop segment start-time (in seconds) of stored function table number *x*. This reports the duration of the direct recorded attack and decay parts of a sound sample, prior to its looped segment. Returns zero (and a warning message) if the sample does not contain loop points.

## Examples

Here is an example of the ftlptim opcode. It uses the files *ftlptim.csd* [examples/ftlptim.csd], and *Church.wav* [examples/Church.wav].

### Example 362. Example of the ftlptim opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o ftlptim.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs  =1

instr 1

itim = ftlptim(1)
print itim
aout loscil3 .8, 40, 1
outs aout, aout

endin
</CsInstruments>
<CsScore>
f 1 0 0 1 "Church.wav" 0 0 0 ;Csound computes tablesize
```

```
i 1 0 5  
e  
</CsScore>  
</CsoundSynthesizer>
```

The audio file “Church.wav” is a looped sample, its output should include lines like these:

```
Base Note : 60 Detune      : 0  
Low  Note : 0 High Note  : 127  
Low  Vel. : 0 High Vel.  : 127  
Gain     : 1 Count       : 1  
mode     : 801  
start    : 4529 end       : 4912 count :0  
mode     : 0  
start    : 0 end         : 0 count   :0
```

## See Also

*ftchnls, filen, ftsr, nsamp*

## Credits

Author: Barry L. Vercoe  
MIT  
Cambridge, Massachussetts  
1997

# ftmorf

ftmorf — Morphs between multiple ftables as specified in a list.

## Description

Uses an index into a table of ftable numbers to morph between adjacent tables in the list. This morphed function is written into the table referenced by *iresfn* on every k-cycle.

## Syntax

```
ftmorf kftndx, iftfn, iresfn
```

## Initialization

*iftfn* -- The table containing the numbers of any existing tables which are used for the morphing.

*iresfn* -- Table number of the morphed function

The length of all the tables in *iftfn* must equal the length of *iresfn*.

## Performance

*kftndx* -- the index into the *iftfn* table.

If *iftfn* contains (6, 4, 6, 8, 7, 4):

- *kftndx*=4 will write the contents of f7 into *iresfn*.
- *kftndx*=4.5 will write the average of the contents of f7 and f4 into *iresfn*.



### Note

*iresfn* is only updated if the morphing index changes its value, if *kftndx* is static, no writing to *iresfn* will occur.

## Examples

Here is an example of the ftmorf opcode. It uses the file *ftmorf.csd* [examples/ftmorf.csd].

### Example 363. Example of the ftmorf opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o ftmorf.wav -W ;; for file output any platform
```

```
</CsOptions>
<CsInstruments>

  sr = 44100
  ksmps = 32
  nchnls = 2
  0dbfs = 1

  instr 1

  kndx line 0, p3, 7
      ftmorf kndx, 1, 2
  asig oscili .8, 440, 2
      outs asig, asig

  endin
</CsInstruments>
<CsScore>

f1 0 8 -2 3 4 5 6 7 8 9 10
f2 0 1024 10 1 /*contents of f2 dont matter */
f3 0 1024 10 1
f4 0 1024 10 0 1
f5 0 1024 10 0 0 1
f6 0 1024 10 0 0 0 1
f7 0 1024 10 0 0 0 0 1
f8 0 1024 10 0 0 0 0 0 1
f9 0 1024 10 0 0 0 0 0 0 1
f10 0 1024 10 1 1 1 1 1 1 1 1

i1 0 15
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: William "Pete" Moss  
University of Texas at Austin  
Austin, Texas USA  
Jan. 2002

New in version 4.18

# ftom

ftom — Convert frequency to midi

## Description

Convert frequency to midi note number, taking global value of A4 into account.

## Syntax

```
imidi ftom ifreq  
kmidi ftom kfreq
```

## Performance

*kfreq / ifreq* -- Frequency

*kmidi / imidi* -- Corresponding midi note number



### Note

Set the global *A4* value at the header to modify the tuning

## Examples

Here is an example of the ftom opcode. It uses the file *mtof-ftom.csd* [examples/mtof-ftom.csd].

### Example 364. Example of the ftom opcode.

```
<CsoundSynthesizer>  
<CsoundOptions>  
</CsoundOptions>  
<CsoundInstruments>  
  
sr = 44100  
ksmps = 128  
nchnls = 2  
odbf = 1.0  
A4  
  
instr 1  
  kfreq = mtof(69)  
  printks2 "midi 69 -> %f\n", kfreq  
  
  kmidi = ftom(442)  
  printks2 "freq 442 -> %f\n", kmidi  
  
  kfreq = mtof(kmidi)  
  printks "midi %f -> %f\n", 1, kmidi, kfreq  
  
  imidi = ftom:i(440)  
  print imidi  
  
  ifreq = mtof:i(60)
```

```
print ifreq

turnoff
endin

</CsInstruments>
<CsScore>
i 1 0 10

</CsScore>
</CsoundSynthesizer>
```

## See Also

*cpsmidim, mtof*

## Credits

By: Eduardo Moguillansky 2017

New in version 6.09

# ftsamplebank

ftsamplebank — Reads a directory for sound files.

## Description

Reads a directory for sound files and loads them to a series of GEN01 function tables.

## Syntax

```
iNumberOfFile ftsamplebank SDirectory, iFirstTableNumber, iSkipTime, iFormat, iChannel,  
kNumberOfFile ftsamplebank SDirectory, kFirstTableNumber, kTrigger, kSkipTime, kFormat, kChannel,
```

## Initialization

*SDirectory* -- a string that identifies the directory to browse for files

*FirstTableNumber* -- this sets the number of the first table into which a soundfile will be loaded

*Trigger* -- updates the tables when kTrigger is 1, only applicable to k-rate version.

*SkipTime* -- begin reading at *skiptime* seconds into the file.

*Format* -- specifies the audio data-file format:

1 - 8-bit signed character	4 - 16-bit short integers
2 - 8-bit A-law bytes	5 - 32-bit long integers
3 - 8-bit U-law bytes	6 - 32-bit floats

*Channel* -- channel number to read in. 0 denotes read all channels.

If *format* = 0 the sample format is taken from the soundfile header, or by default from the CSound *-o* command-line flag.

## Performance

*iNumberOfFile* -- the number of tables that have been created

*kNumberOfFile* -- the number of tables that have been created



### Note

Loading a lot of sound files into function tables at k-rate may cause some audio dropouts.

## Example

This example shows `ftsamplebank` browsing for samples in a fixed location. It loads all samples it finds into GEN01 function tables and then plays them back in a sequence, one every second. It uses the file `ftsamplebank.csd` [examples/ftsamplebank.csd].

**Example 365. Example of the ftsamplebank opcode.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o diskin.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
Odbfs = 1

;load all samples in a given directory into function tables and play them using instrument 1000
instr 1
  iFirstTableNumber = 60;
  iFileCount init 0
  iNumberOfFiles ftsamplebank "./examples/", iFirstTableNumber, 0, 4, 1

  until iFileCount>=iNumberOfFiles do
event_i "i", 1000, iFileCount, 1, iFirstTableNumber+iFileCount
    iFileCount = iFileCount+1
  enduntil
endin

instr 1000
  iTable = p4
  aOut loscil3 1, 1, iTable, 1, 0;
outs aOut, aOut
endin

</CsInstruments>
<CsScore>
i1 0 100
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Rory Walsh  
2015

# ftsave

ftsave — Save a set of previously-allocated tables to a file.

## Description

Save a set of previously-allocated tables to a file.

## Syntax

```
ftsave "filename", iflag, ifn1 [, ifn2] [...]
```

## Initialization

*"filename"* -- A quoted string containing the name of the file to save.

*iflag* -- Type of the file to save. (0 = binary file, Non-zero = text file)

*ifn1, ifn2, ...* -- Numbers of tables to save.

## Performance

*ftsave* saves a list of tables to a file. The file's format can be binary or text.



### Warning

The file's format is not compatible with a WAV-file and is not endian-safe.

## Examples

Here is an example of the ftsave opcode. It uses the file *ftsave.csd* [examples/ftsave.csd].

### Example 366. Example of the ftsave opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o ftsave.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1
```

```
; Table #1, make a sine wave using the GEN10 routine.
gitmp1 ftgen 1, 0, 32768, 10, 1
; Table #2, create an empty table.
gitmp2 ftgen 2, 0, 32768, 7, 0, 32768, 0

; Instrument #1 - a basic oscillator.
instr 1
  kamp = 20000
  kcps = 440
  ; Use Table #1.
  ifn = 1

  a1 oscil kamp, kcps, ifn
  out a1
endin

; Instrument #2 - Load Table #1 into Table #2.
instr 2
  ; Save Table #1 to a file called "table1.ftsave".
  ftsave "table1.ftsave", 0, 1

  ; Load the "table1.ftsave" file into Table #2.
  ftload "table1.ftsave", 0, 2

  kamp = 20000
  kcps = 440
  ; Use Table #2, it should contain Table #1's sine wave now.
  ifn = 2

  a1 oscil kamp, kcps, ifn
  out a1
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for 1 second.
i 1 0 1
; Play Instrument #2 for 1 second.
i 2 2 1
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*ftloadk, ftload, ftsavek*

## Credits

Author: Gabriel Maldonado

Example written by Kevin Conder.

New in version 4.21

# ftsavek

`ftsavek` — Save a set of previously-allocated tables to a file.

## Description

Save a set of previously-allocated tables to a file.

## Syntax

```
ftsavek "filename", ktrig, iflag, ifn1 [, ifn2] [...]
```

## Initialization

*"filename"* -- A quoted string containing the name of the file to save.

*iflag* -- Type of the file to save. (0 = binary file, Non-zero = text file)

*ifn1, ifn2, ...* -- Numbers of tables to save.

## Performance

*ktrig* -- The trigger signal. Save the file each time it is non-zero.

*ftsavek* saves a list of tables to a file. The file's format can be binary or text. Unlike *ftsave*, the saving operation can be repeated numerous times within the same note by using a trigger signal.



### Warning

The file's format is not compatible with a WAV-file and is not endian-safe.

## See Also

*ftloadk, ftload, ftsave*

## Credits

Author: Gabriel Maldonado

New in version 4.21

# ftsr

ftsr — Returns the sampling-rate of a stored function table.

## Description

Returns the sampling-rate of a stored function table.

## Syntax

```
ftsr(x) (init-rate args only)
```

## Performance

Returns the sampling-rate of a *GEN01* generated table. The sampling-rate is determined from the header of the original file. If the original file has no header or the table was not created by these *GEN01*, *ftsr* returns 0. New in Csound version 3.49.

## Examples

Here is an example of the *ftsr* opcode. It uses the file *ftsr.csd* [examples/ftsr.csd].

### Example 367. Example of the *ftsr* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o ftsr.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

itab = p4
isr = ftsr(itab)
prints "sampling-rate of table number %d = %d\n", itab, isr

endin
</CsInstruments>
<CsScore>
f 1 0 0 1 "kickroll.wav" 0 0 0 ;stereo file
f 2 0 0 1 "ahh.aiff" 0 0 0 ;& different formats
f 3 0 0 1 "beats.mp3" 0 0 0
f 4 0 0 1 "beats.ogg" 0 0 0
```

```
i 1 0 1 1
i 1 + 1 2
i 1 + 1 3
i 1 + 1 4
e
</CsScore>
</CsoundSynthesizer>
```

its output should a line like these:

```
sampling-rate of table number 1 = 44100
sampling-rate of table number 2 = 22050
sampling-rate of table number 3 = 44100
sampling-rate of table number 4 = 44100
```

## See Also

*ftchnls, flen, flptim, nsamp*

## Credits

Author: Gabriel Maldonado  
Italy  
October 1998

# gain

gain — Adjusts the amplitude audio signal according to a root-mean-square value.

## Description

Adjusts the amplitude audio signal according to a root-mean-square value.

## Syntax

```
ares gain asig, krms [, ihp] [, iskip]
```

## Initialization

*ihp* (optional, default=10) -- half-power point (in Hz) of a special internal low-pass filter. The default value is 10.

*iskip* (optional, default=0) -- initial disposition of internal data space (see *reson*). The default value is 0.

## Performance

*asig* -- input audio signal

*gain* provides an amplitude modification of *asig* so that the output *ares* has rms power equal to *krms*. *rms* and *gain* used together (and given matching *ihp* values) will provide the same effect as *balance*.

## Examples

Here is an example of the gain opcode. It uses the file *gain.csd* [examples/gain.csd].

### Example 368. Example of the gain opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gain.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

  asrc buzz .8, 440, sr/440, 1 ; band-limited pulse train.
  a1 reson asrc, 1000, 100 ; Sent through
```

```
a2 reson a1, 3000, 500      ; 2 filters
krms rms  asrc            ; then balanced
afin gain a2, krms        ; with source
      outs afin, afin
endin
```

```
</CsInstruments>
<CsScore>
; sine wave.
f 1 0 16384 10 1

i 1 0 2
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*balance, rms*

# gainslider

`gainslider` — An implementation of a logarithmic gain curve which is similar to the `gainslider~` object from Cycling 74 Max / MSP.

## Description

This opcode is intended for use to multiply by an audio signal to give a console mixer like feel. There is no bounds in the source code so you can for example give higher than 127 values for extra amplitude but possibly clipped audio.

## Syntax

```
kout gainslider kindex
```

## Performance

*kindex* -- Index value. Nominal range from 0-127. For example a range of 0-152 will give you a range from  $-\infty$  to +18.0 dB.

*kout* -- Scaled output.

## Examples

Here is an example of the `gainslider` opcode. It uses the file `gainslider.csd` [examples/gainslider.csd].

### Example 369. Example of the gainslider opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  Silent
-odac        -iadc     -d      ;;realtime output
</CsOptions>
<CsInstruments>

sr = 48000
ksmps = 100
nchnls = 2

instr 1 ; gainslider test

; uncomment for realtime midi
;kmod ctrl7 1, 1, 0, 127

; uncomment for non realtime
km0d phasor 1/10
kmod scale km0d, 127, 0

kout gainslider kmod

printks "kmod = %f  kout = %f\\n", 0.1, kmod, kout
```

```
aout diskin2 "fox.wav", 1, 0, 1  
  
aout = aout*kout  
  
outs aout, aout  
  
    endin  
  
</CsInstruments>  
<CsScore>  
i 0 30  
e  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*scale, logcurve, expcurve*

## Credits

Author: David Akbari  
October  
2006

# gauss

gauss — Gaussian distribution random number generator.

## Description

Gaussian distribution random number generator. This is an x-class noise generator.

## Syntax

```
ares gauss krange
ires gauss krange
kres gauss krange
```

## Performance

*krange* -- the range of the random numbers (*-krange* to *+krange*). Outputs both positive and negative numbers.

*gauss* returns random numbers following a normal distribution centered around 0.0 ( $\mu = 0.0$ ) with a variance (sigma) of *krange* / 3.83. Thus more than 99.99% of the random values generated are in the range *-krange* to *+krange*. If a mean value different of 0.0 is desired, this mean value has to be added to the generated numbers (see example below).

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the gauss opcode. It uses the file *gauss.csd* [examples/gauss.csd].

### Example 370. Example of the gauss opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>

<CsOptions>
  -d -o dac
</CsOptions>

<CsInstruments>
instr 1
  irange = p4
  imu    = p5
  isamples = p6
  indx   = 0
```

```

icount = 1
ix      = 0.0
ix2     = 0.0

loop:
i1      gauss  irange
i1      =      i1 + imu
ix      =      ix + i1
ix2     =      ix2 + i1*i1
if i1 >= -(irange+imu) && i1 <= (irange+imu) then
  icount = icount+1
endif

      loop_lt indx, 1, isamples, loop

imean   =      ix / isamples                ;mean value
istd    =      sqrt(ix2/isamples - imean*imean) ;standard deviation
prints  "mean = %3.3f, std = %3.3f, ", imean, istd
prints  "samples inside the given range: %3.3f%\n", icount*100.0/isamples
endin
</CsInstruments>
<CsScore>
i 1 0 0.1 1.0 0 100000 ; range = 1, mu = 0.0, sigma = 1/3.83 = 0.261
i 1 0.1 0.1 3.83 0 100000 ; range = 3.83, mu = 0.0, sigma = 1
i 1 0.2 0.1 5.745 2.7 100000 ; range = 5.745, mu = 2.7, sigma = 5.745/3.83 = 1.5
</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like this:

```

mean = 0.000, std = 0.260, samples inside the given range: 99.993%
mean = 0.005, std = 0.999, samples inside the given range: 99.998%
mean = 2.700, std = 1.497, samples inside the given range: 100.000%

```

## See Also

*seed, betarand, bexprnd, cauchy, exprand, linrand, pcauchy, poisson, trirand, unirand, weibull*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

Precisions about mu and sigma added by François Pinot after a discussion with Joachim Heintz on the Csound List, December 2010.

Example written by François Pinot, adapted from a csd file by Joachim Heintz, December 2010.

Existed in 3.30

# gaussi

gaussi — Gaussian distribution random number generator with interpolation.

## Description

Gaussian distribution random number generator with controlled interpolation between values. This is an x-class noise generator.

## Syntax

```
ares gaussi krange, xamp, xcps
```

```
ires gaussi krange, xamp, xcps
```

```
kres gaussi krange, xamp, xcps
```

## Performance

*krange* -- the range of the random numbers (*-krange* to *+krange*). Outputs both positive and negative numbers.

*gauss* returns random numbers following a normal distribution centered around 0.0 ( $\mu = 0.0$ ) with a variance (sigma) of *krange* / 3.83. Thus more than 99.99% of the random values generated are in the range *-krange* to *+krange*. If a mean value different of 0.0 is desired, this mean value has to be added to the generated numbers (see example below).

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

*xamp* -- range over which random numbers are distributed.

*xcps* -- the frequency which new random numbers are generated.

## Examples

Here is an example of the *gaussi* opcode. It uses the file *gaussi.csd* [examples/gaussi.csd].

### Example 371. Example of the *gaussi* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
```

```
; -o exprand.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
klambda gaussi 100, 1, 3
printk2 klambda ; look
aout oscili 0.8, 440+klambda, 1 ; & listen
outs aout, aout
endin

</CsInstruments>
<CsScore>
; sine wave
f 1 0 16384 10 1

i 1 0 4
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*gauss*

## Credits

Author: John ffitth  
Bath  
May 2011  
New in version 5.14

# gausstrig

gausstrig — Random impulses around a certain frequency.

## Description

Generates random impulses around a certain frequency.

## Syntax

```
ares gausstrig kamp, kcps, kdev [, imode] [, ifrst1]
```

```
kres gausstrig kamp, kcps, kdev [, imode] [, ifrst1]
```

## Initialization

*imode* (optional, default=0) -- *imode* > 0 means better frequency modulation. If the frequency changes, the delay before the next impulse is calculated again. With the default mode we have the classic behavior of the GaussTrig ugen in SuperCollider, where the frequency modulation is bypassed during the delay time that precedes the next impulse.

*ifrst1* (optional, default=0) -- *ifrst1* > 0 changes the original behavior of the GaussTrig ugen. By default, it always generates an impulse in the very beginning. Otherwise, the first impulse appearance is random and depends on *kcps* and *kdev* parameters.

## Performance

*kamp* -- amplitude.

*kcps* -- the mean frequency over which random impulses are distributed.

*kdev* -- random deviation from mean ( $0 \leq \text{dev} < 1$ ).

## Examples

Here is an example of the gausstrig opcode. It uses the file *gausstrig.csd* [examples/gausstrig.csd].

### Example 372. Example of the gausstrig opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o oscil.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kdev line 0, p3, 0.9
seed 20120125
aimp gausstrig 0.5, 10, kdev
aenv filter2 aimp, 1, 1, 0.993, 0.993
anoi fractalnoise 0.2, 1.7
al = anoi*aenv
ar delay al, 0.02
outs al, ar

endin
</CsInstruments>
<CsScore>
i1 0 10
e
</CsScore>
</CsoundSynthesizer>

```

Here is an example of the `gaustrig` opcode with `imode = 1`. It uses the file `gaustrig-2.csd` [examples/gaustrig-2.csd].

### Example 373. Example of the `gaustrig` opcode with `imode = 1`.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o oscil.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kfrq0 oscil 8, 0.25, 1
ktrig metro 1
kfrq samphold kfrq0+8, ktrig
seed 20120125
aimp gausstrig 0.5, kfrq, 0.5, 1
aenv filter2 aimp, 1, 1, 0.993, 0.993
anoi fractalnoise 0.2, 1.7
al = anoi*aenv
ar delay al, 0.02
outs al, ar

endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1
i1 0 16
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*dust dust2 mpulse*

## Credits

Based on Bhub Rainey's GaussTrig ugen (SuperCollider)

Author: Tito Latini

January 2012

Modification: Gleb Rogozinsky

March 2015

New in Csound version 5.16

# gbuzz

*gbuzz* — Output is a set of harmonically related cosine partials.

## Description

Output is a set of harmonically related cosine partials.

## Syntax

```
ares gbuzz xamp, xcps, knh, klh, kmul, ifn [, iphs]
```

## Initialization

*ifn* -- table number of a stored function containing a cosine wave. A large table of at least 8192 points is recommended.

*iphs* (optional, default=0) -- initial phase of the fundamental frequency, expressed as a fraction of a cycle (0 to 1). A negative value will cause phase initialization to be skipped. The default value is zero

## Performance

The buzz units generate an additive set of harmonically related cosine partials of fundamental frequency *xcps*, and whose amplitudes are scaled so their summation peak equals *xamp*. The selection and strength of partials is determined by the following control parameters:

*knh* -- total number of harmonics requested. If *knh* is negative, the absolute value is used. If *knh* is zero, a value of 1 is used.

*klh* -- lowest harmonic present. Can be positive, zero or negative. In *gbuzz* the set of partials can begin at any partial number and proceeds upwards; if *klh* is negative, all partials below zero will reflect as positive partials without phase change (since cosine is an even function), and will add constructively to any positive partials in the set.

*kmul* -- specifies the multiplier in the series of amplitude coefficients. This is a power series: if the *klh*th partial has a strength coefficient of *A*, the (*klh* + *n*)th partial will have a coefficient of  $A * (kmul ** n)$ , i.e. strength values trace an exponential curve. *kmul* may be positive, zero or negative, and is not restricted to integers.

*buzz* and *gbuzz* are useful as complex sound sources in subtractive synthesis. *buzz* is a special case of the more general *gbuzz* in which *klh* = *kmul* = 1; it thus produces a set of *knh* equal-strength harmonic partials, beginning with the fundamental. (This is a band-limited pulse train; if the partials extend to the Nyquist, i.e.  $knh = \text{int}(sr / 2 / \text{fundamental freq.})$ , the result is a real pulse train of amplitude *xamp*.)

Although both *knh* and *klh* may be varied during performance, their internal values are necessarily integer and may cause “pops” due to discontinuities in the output. *kmul*, however, can be varied during performance to good effect. *gbuzz* can be amplitude- and/or frequency-modulated by either control or audio signals.

N.B. This unit has its analog in *GENII*, in which the same set of cosines can be stored in a function table for sampling by an oscillator. Although computationally more efficient, the stored pulse train has a fixed spectral content, not a time-varying one as above.

## Examples

Here is an example of the `gbuzz` opcode. It uses the file `gbuzz.csd` [examples/gbuzz.csd].

### Example 374. Example of the `gbuzz` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gbuzz.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs   =1

instr 1

kcps = 220
knh = p4 ;total no. of harmonics
klh = p5 ;lowest harmonic
kmul line 0, p3, 1 ;increase amplitude of
;higer partials
asig gbuzz .6, kcps, knh, klh, kmul, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
; a cosine wave
f 1 0 16384 11 1

i 1 0 3 3 1 ;3 harmonics, lowest harmonic=1
i 1 + 3 30 1 ;30 harmonics, lowest harmonic=1
i 1 + 3 3 2 ;3 harmonics, lowest harmonic=3
i 1 + 3 30 2 ;30 harmonics, lowest harmonic=3
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*buzz*

## Credits

September 2003. Thanks to Kanata Motohashi for correcting the mentions of the *kmul* parameter.

# genarray

genarray — Generate a vector with an arithmetic sequence.

## Description

Generate a vector (one-dimensional k-rate or i-rate array) with an arithmetic sequence.

## Syntax

```
karray genarray kstart, kens[, inc]
iarray genarray istart, iens[, inc]
```

## Initialization

*istart* -- value to place in first element.

*iend* -- last value to place in array.

*inc* -- amount to add to previous value (default 1).

## Examples

Here is an example of the genarray opcode. It uses the file *genarray.csd* [examples/genarray.csd].

### Example 375. Example of the genarray opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
odbfs = 1

instr 1

;create and fill two arrays
iArr1[] genarray 1, 6
iArr2[] genarray 1, 6, 2/3

;print the content of iArr1
printf "%s", 1, "iArr1: start=1, end=6, step=default\n"
kndx = 0
  until kndx == lenarray(iArr1) do
    printf "iArr[%d] = %f\n", kndx+1, kndx, iArr1[kndx]
    kndx += 1
  od
```

```
;print the content of iArr2
printf "%s", 1, "iArr2: start=1, end=6, step=2/3\n"
kndx = 0
  until kndx == lenarray(iArr2) do
    printf "iArr[%d] = %f\n", kndx+1, kndx, iArr2[kndx]
    kndx += 1
  od

  turnoff
endin

</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsSoundSynthesizer>
```

## See Also

Vectorial opcodes

## Credits

Author: John ffitch  
Codemist Ltd  
2013

New in version 6.00

# genarray\_i

genarray\_i — Generate a vector with an arithmetic sequence.

## Description

Generate a vector (one-dimensional k-rate) with an arithmetic sequence at initialisation time.

## Syntax

```
karray genarray_i istart, iend [,inc]
```

## Initialization

*istart* -- value to place in first element.

*iend* -- last value to place in array.

*inc* -- amount to add to previous value (default 1).

## Examples

Here is an example of the genarray\_i opcode. It uses the file *genarray\_i.csd* [examples/genarray\_i.csd].

### Example 376. Example of the genarray\_i opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
odbfs = 1

instr 1

;create and fill two arrays
kArr1[] genarray_i 1, 6
kArr2[] genarray_i 1, 6, 2/3

;print the content of kArr1
printf "%s", 1, "kArr1: start=1, end=6, step=default\n"
kndx = 0
until kndx == lenarray(kArr1) do
printf "kArr[%d] = %f\n", kndx+1, kndx, kArr1[kndx]
kndx += 1
od

;print the content of kArr2
```

```
        printf "%s", 1, "kArr2: start=1, end=6, step=2/3\n"
kndx   = 0
    until kndx == lenarray(kArr2) do
        printf "kArr[%d] = %f\n", kndx+1, kndx, kArr2[kndx]
kndx   += 1
    od

        turnoff
    endin

</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsoundSynthesizer>
```

## See Also

Vectorial opcodes *fillarray*

## Credits

Author: John ffitth  
Codemist Ltd  
2013

New in version 6.00

# gendy

`gendy` — Dynamic stochastic approach to waveform synthesis conceived by Iannis Xenakis.

## Description

Implementation of the *Génération Dynamique Stochastique* (GENDYN), a dynamic stochastic approach to waveform synthesis conceived by Iannis Xenakis.

## Syntax

```
ares gendy kamp, kampdist, kdurdist, kadpar, kddpar, kminfreq, kmaxfreq, \  
          kampscl, kdurscl [, initcps] [, knum]  
  
kres gendy kamp, kampdist, kdurdist, kadpar, kddpar, kminfreq, kmaxfreq, \  
          kampscl, kdurscl [, initcps] [, knum]
```

## Initialization

`initcps` (optional, default=12) -- max number of control points.

## Performance

`kamp` -- amplitude.

`kampdist` -- choice of probability distribution for the next perturbation of the amplitude of a control point. The valid distributions are:

- 0 - LINEAR
- 1 - CAUCHY
- 2 - LOGIST
- 3 - HYPERBCOS
- 4 - ARCSINE
- 5 - EXPON
- 6 - SINUS (external k-rate signal)

If `kampdist=6`, the user can use an external k-rate signal through `kadpar`.

`kdurdist` -- choice of distribution for the perturbation of the current inter control point duration. See `kampdist` for the valid distributions. If `kdurdist=6`, the user can use an external k-rate signal through `kddpar`.

`kadpar` -- parameter for the `kampdist` distribution. Should be in the range of 0.0001 to 1.

`kddpar` -- parameter for the `kdurdist` distribution. Should be in the range of 0.0001 to 1.

`kminfreq` -- minimum allowed frequency of oscillation.

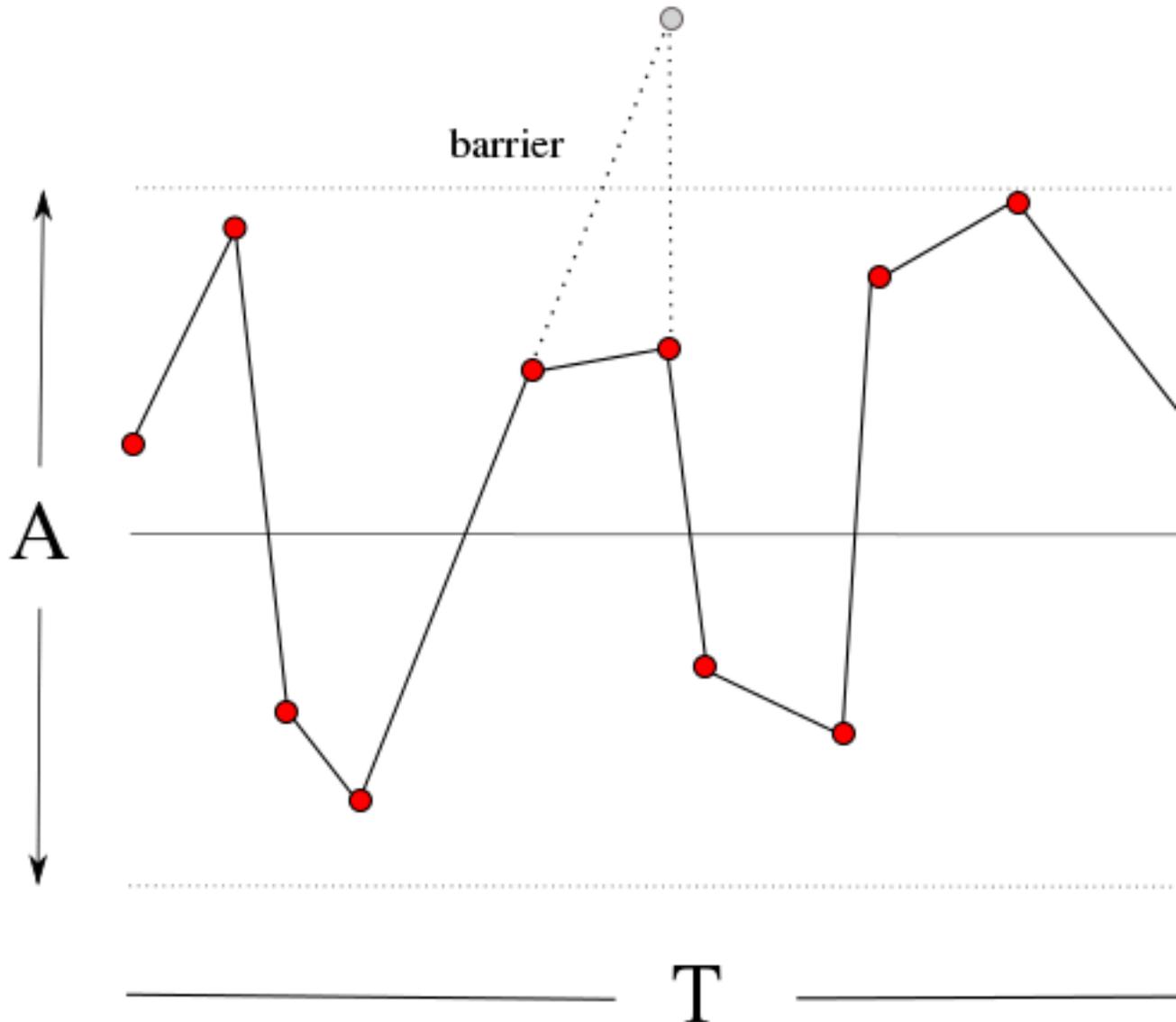
*kmaxfreq* -- maximum allowed frequency of oscillation.

*kampscl* -- multiplier for the distribution's delta value for amplitude (1.0 is full range).

*kdurscl* -- multiplier for the distribution's delta value for duration.

*knum* (optional, default=*initcps*) -- current number of utilized control points.

The waveform is generated by  $knum - 1$  segments and is repeated in the time. The vertexes (control points) are moved according to a stochastic action and they are limited within the boundaries of a mirror formed by an amplitude barrier and a time barrier.



A repetition of the generated waveform with  $knum=12$ .

## Examples

Here is an example of the *gendy* opcode. It uses the file *gendy.csd* [examples/*gendy.csd*].

**Example 377. Example of the gendy opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o oscil.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

aout gendy 0.7, 1, 1, 1, 1, 20, 1000, 0.5, 0.5
outs aout, aout

endin
</CsInstruments>
<CsScore>
i1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

Here is an example of the gendy opcode with some modulations. It uses the file *gendy-2.csd* [examples/gendy-2.csd].

**Example 378. Example of the gendy opcode with some modulations.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o oscil.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kenv expseg 0.01, p3*0.1, 1, p3*0.8, 1, p3*0.1, 0.001
kosc oscil 0.1, 3/p3, 1
seed 20120123
kdis bexprnd kosc
knum linseg 3, p3*0.75, 10, p3*0.20, 12, p3*0.05, 5
```

```
asig gendy 0.2, kosc*60, 6, 0.7, kdis, 500*kenv, 4800, 0.23, 0.3, 12, knum
aflt resonz asig, 1400, 400
aout comb kenv*aflt*0.1, 0.9, 0.1
outs aout, aout

endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1 0 .8 0 0 .3 0 0 0 .1

i1 0 20
e
</CsScore>
</CsoundSynthesizer>
```

## References

1. Formalized Music (1992, Stuyvesant, NY: Pendragon Press), pp. 246 - 254, 289 - 322.

## See Also

*gendyc gendyx*

## Credits

Based on Nick Collins's Gendy1 ugen (SuperCollider)

Author: Tito Latini

January 2012

New in Csound version 5.16

# gendyc

gendyc — Dynamic stochastic approach to waveform synthesis using cubic interpolation.

## Description

Implementation with cubic interpolation of the *Génération Dynamique Stochastique* (GENDYN), a dynamic stochastic approach to waveform synthesis conceived by Iannis Xenakis.

## Syntax

```
ares gendyc kamp, kampdist, kdurdist, kadpar, kddpar, kminfreq, kmaxfreq, \  
kampscl, kdurscl [, initcps] [, knum]  
  
kres gendyc kamp, kampdist, kdurdist, kadpar, kddpar, kminfreq, kmaxfreq, \  
kampscl, kdurscl [, initcps] [, knum]
```

## Initialization

*initcps* (optional, default=12) -- max number of control points.

## Performance

*kamp* -- amplitude.

*kampdist* -- choice of probability distribution for the next perturbation of the amplitude of a control point. The valid distributions are:

- 0 - LINEAR
- 1 - CAUCHY
- 2 - LOGIST
- 3 - HYPERBCOS
- 4 - ARCSINE
- 5 - EXPON
- 6 - SINUS (external k-rate signal)

If *kampdist*=6, the user can use an external k-rate signal through *kadpar*.

*kdurdist* -- choice of distribution for the perturbation of the current inter control point duration. See *kampdist* for the valid distributions. If *kdurdist*=6, the user can use an external k-rate signal through *kddpar*.

*kadpar* -- parameter for the *kampdist* distribution. Should be in the range of 0.0001 to 1.

*kddpar* -- parameter for the *kdurdist* distribution. Should be in the range of 0.0001 to 1.

*kminfreq* -- minimum allowed frequency of oscillation.

*kmaxfreq* -- maximum allowed frequency of oscillation.

*kampscl* -- multiplier for the distribution's delta value for amplitude (1.0 is full range).

*kdurscl* -- multiplier for the distribution's delta value for duration.

*knum* (optional, default=*initcps*) -- current number of utilized control points.

The waveform is generated by *knum* - 1 segments and is repeated in the time. The vertexes (control points) are moved according to a stochastic action and they are limited within the boundaries of a mirror formed by an amplitude barrier and a time barrier.

## Examples

Here is an example of the *gendyc* opcode. It uses the file *gendyc.csd* [examples/gendyc.csd].

### Example 379. Example of the *gendyc* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o oscil.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

aout gendyc 0.5, 1, 1, 1, 1, 220, 440, 0.5, 0.5
outs aout, aout

endin
</CsInstruments>
<CsScore>
i1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

## References

1. Formalized Music (1992, Stuyvesant, NY: Pendragon Press), pp. 246 - 254, 289 - 322.

## See Also

*gendy gendyx*

## Credits

Based on Nick Collins's *Gendy1* and Bhub Rainey's *Gendy4* (SuperCollider)

Author: Tito Latini  
January 2012

New in Csound version 5.16

# gendyx

*gendyx* — Variation of the dynamic stochastic approach to waveform synthesis conceived by Iannis Xenakis.

## Description

*gendyx* (*gendy* eXtended) is an implementation of the *Génération Dynamique Stochastique* (GENDYN), a dynamic stochastic approach to waveform synthesis conceived by Iannis Xenakis, using curves instead of segments.

## Syntax

```
ares gendyx kamp, kampdist, kdurdist, kadpar, kddpar, kminfreq, kmaxfreq, \  
             kampscl, kdurscl, kcurveup, kcurvedown [, initcps] [, knum]  
  
kres gendyx kamp, kampdist, kdurdist, kadpar, kddpar, kminfreq, kmaxfreq, \  
             kampscl, kdurscl, kcurveup, kcurvedown [, initcps] [, knum]
```

## Initialization

*initcps* (optional, default=12) -- max number of control points.

## Performance

*kamp* -- amplitude.

*kampdist* -- choice of probability distribution for the next perturbation of the amplitude of a control point. The valid distributions are:

- 0 - LINEAR
- 1 - CAUCHY
- 2 - LOGIST
- 3 - HYPERBCOS
- 4 - ARCSINE
- 5 - EXPON
- 6 - SINUS (external k-rate signal)

If *kampdist*=6, the user can use an external k-rate signal through *kadpar*.

*kdurdist* -- choice of distribution for the perturbation of the current inter control point duration. See *kampdist* for the valid distributions. If *kdurdist*=6, the user can use an external k-rate signal through *kddpar*.

*kadpar* -- parameter for the *kampdist* distribution. Should be in the range of 0.0001 to 1.

*kddpar* -- parameter for the *kdurdist* distribution. Should be in the range of 0.0001 to 1.

*kminfreq* -- minimum allowed frequency of oscillation.

*kmaxfreq* -- maximum allowed frequency of oscillation.

*kampscl* -- multiplier for the distribution's delta value for amplitude (1.0 is full range).

*kdurscl* -- multiplier for the distribution's delta value for duration.

*kcurveup* -- controls the curve for the increasing amplitudes between two points; it has to be non negative.

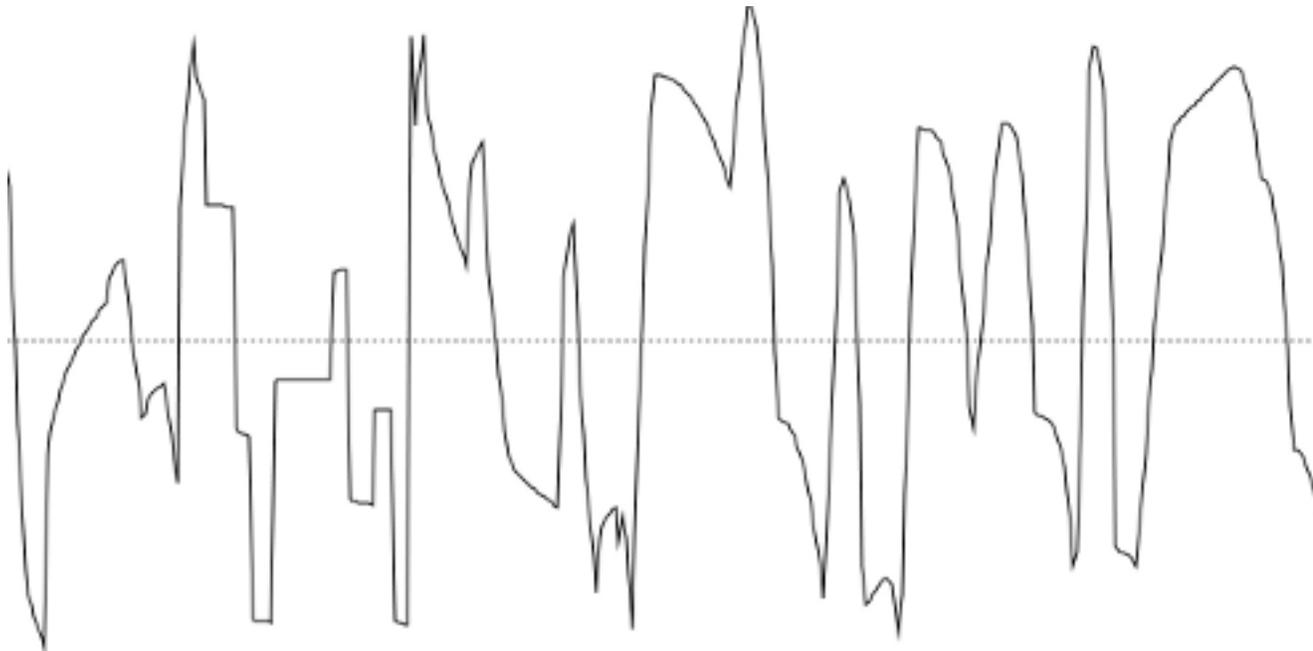
- 0: step function (like a sample & hold)
- <1: concave
- 1: linear (like gendy)
- >1: convex

*kcuredown* -- controls the curve for the decreasing amplitudes between two points; it has to be non negative.

- 0: step function
- <1: convex
- 1: linear
- >1: concave

*knum* (optional, default=*initcps*) -- current number of utilized control points.

The waveform is generated by *knum* - 1 curves and is repeated in the time. The vertexes (control points) are moved according to a stochastic action and they are limited within the boundaries of a mirror formed by an amplitude barrier and a time barrier.



Extract of a waveform generated with *gendyx*.

## Examples

Here is an example of the `gendyx` opcode. It uses the file `gendyx.csd` [examples/gendyx.csd].

### Example 380. Example of the `gendyx` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o oscil.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

aout gendyx 0.7, 1, 1, 1, 1, 20, 1000, 0.5, 0.5, 4, 0.13
outs aout, aout

endin
</CsInstruments>
<CsScore>
i1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

Here is an example of the `gendyx` opcode with some modulations. It uses the file `gendyx-2.csd` [examples/gendyx-2.csd].

### Example 381. Example of the `gendyx` opcode with some modulations.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o oscil.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
```

```
kenv expseg 0.001, p3*0.05, 0.5, p3*0.9, 0.5, p3*0.05, 0.001
kc1 linseg 1, p3/2, 12, p3/2, 3
kc2 random 0, 4
seed 20120124
asig gendyx kenv, 1, 3, 0.7, 0.8, 120, 4300, 0.2, 0.7, kc1, kc2, 12, kc1
aout dcblock asig
outs aout, aout
endin
</CsInstruments>
<CsScore>
i1 0 20
e
</CsScore>
</CsoundSynthesizer>
```

## References

1. Formalized Music (1992, Stuyvesant, NY: Pendragon Press), pp. 246 - 254, 289 - 322.

## See Also

*gendy gendyc*

## Credits

Variation of the Nick Collins's Gendy1 ugen (SuperCollider)

Author: Tito Latini

January 2012

New in Csound version 5.16

# getcfg

getcfg — Return Csound settings.

## Description

Return various configuration settings in Svalue as a string at init time.

## Syntax

Svalue **getcfg** iopt

## Initialization

*iopt* -- The parameter to be returned, can be one of:

- 1: the maximum length of string variables in characters; this is at least the value of the `--max_str_len` command line option - 1.



### Note

In Csound6 there is no maximum string length so the returned value is meaningless

- 2: the input sound file name (-i), or empty if there is no input file
- 3: the output sound file name (-o), or empty if there is no output file
- 4: return "1" if real time audio input or output is being used, and "0" otherwise
- 5: return "1" if running in beat mode (-t command line option), and "0" otherwise
- 6: the host operating system name
- 7: return "1" if a callback function for the `chnrecv` and `chnsend` opcodes has been set, and "0" otherwise (which means these opcodes do nothing)

## Examples

Here is an example of the `getcfg` opcode. It uses the file `getcfg.csd` [examples/getcfg.csd].

### Example 382. Example of the `getcfg` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
-iadc      ;;uncomment -iadc if realtime audio input is needed too
</CsOptions>
<CsInstruments>
```

```

sr = 44100
ksmps = 32
nchnls = 2
odbfs = 1

instr 1

S1 getcfg 1 ; ++max_str_len
S2 getcfg 2 ; -i
S3 getcfg 3 ; -o
S4 getcfg 4 ; RTaudio
S5 getcfg 5 ; -t
S6 getcfg 6 ; os system host
S7 getcfg 7 ; callback

prints "-----"
prints "\nMax string len : "
prints S1
prints "\nInput file name (-i) : "
prints S2
prints "\nOutput file name (-o) : "
prints S3
prints "\nRTaudio (-odac) : "
prints S4
prints "\nBeat mode (-t)? : "
prints S5
prints "\nHost Op. Sys. : "
prints S6
prints "\nCallback ? : "
prints S7
prints "\n"
prints "-----\n"

endin

</CsInstruments>
<CsScore>
i 1 0 0
e
</CsScore>
</CsoundSynthesizer>

```

The output should include lines like these:

```

-----
Max string len : 255
Input file name (-i) : adc
Output file name (-o) : dac
RTaudio (-odac) : 1
Beat mode (-t)? : 0
Host Op. Sys. : Linux
Callback ? : 0
-----

```

## Credits

Author: Istvan Varga  
2006

New in version 5.02

# getcol

getcol — Gets a given column from a 2-dimensional array as a vector.

## Description

Gets a given column from a 2-dimensional array. The output is an 1-d array with the contents of the requested column

## Syntax

```
kout[] getcolkin[],kcol
```

## Performance

*kout[]* -- output array containing the extracted column. It will be created if it does not exist.

*kin[]* -- input 2-dimensional array.

*kcol* -- column to be extracted.

## Examples

Here is an example of the getcol opcode. It uses the file *getcol.csd* [examples/getcol.csd].

### Example 383. Example of the getcol opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
instr 1
  kcnt init 0
  kArr[] init 3,3
  kArr[] fillarray 0,1,2,0,1,2,0,1,2
  while kcnt < 3 do
    kVec[] getcol kArr,kcnt
    printf "column %d: %d %d %d\n",kcnt+1,kcnt,kVec[0],kVec[1],kVec[2]
    kcnt += 1
  od
endin
</CsInstruments>
<CsScore>
i1 0 0.1
</CsScore>
</CsoundSynthesizer>
```

## See Also

Vectorial opcodes array opcodes

## Credits

Author: Victor Lazzarini  
NUI Maynooth  
2014

New in version 6.04

# getftargs

getftargs — Fill a string variable with the arguments used to create a function table at k-rate.

## Description

*getftargs* writes the arguments used to create a function table to a string variable. *getftargs* runs both at initialization and performance time.

## Syntax

```
Sdst getftargs iftno, ktrig
```

## Initialization

*iftno* -- Number of the table whose arguments we want to request.

## Performance

*Sdst* -- output string variable.

*ktrig* -- trigger signal, should be valid at i-time. The output string variable is filled at initialization time if *ktrig* is positive, and at performance time whenever *ktrig* is both positive and different from the previous value. Use a constant value of 1 to print once at note initialization.

## Examples

Here is an example of the *getftargs* opcode. It uses the file *getftargs.csd* [examples/getftargs.csd].

### Example 384. Example of the *getftargs* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o getftargs.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
Sargs getftargs 1, 1
puts Sargs, 1
endin
```

```
</CsInstruments>
<CsScore>
f 1 0 1024 "quadbezier" 0 0 0.5 200 0.8 450 0.33 600 0.1 800 0.4 1024 0
i 1 0 1
e
</CsScore>
</CsoundSynthesizer>
```

Its output should include this line:

```
0 0 0.5 200 0.8 450 0.33 600 0.1 800 0.4 1024 0
```

## See also

*ftgen*, *filen* and *sprintf*

## Credits

Written by Guillermo Senna

2016

# getrow

getrow — Gets a given row from a 2-dimensional array as a vector.

## Description

Gets a given row from a 2-dimensional array. The output is an 1-d array with the contents of the requested row

## Syntax

```
kout[] getrowkin[],krow
```

## Performance

*kout[]* -- output array containing the extracted row. It will be created if it does not exist.

*kin[]* -- input 2-dimensional array.

*krow* -- row to be extracted.

## Examples

Here is an example of the getrow opcode. It uses the file *rfft.csd* [examples/rfft.csd].

### Example 385. Example of the getrow opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>

<CsOptions>
-d -o dac
</CsOptions>

<CsInstruments>
;ksmps needs to be an integer div of hopsize
ksmps = 64

instr 1

  ihopsize = 256 ; hopsize
  ifftsize = 1024 ; FFT size
  iolaps = ifftsize/ihopsize ; overlaps
  ibw = sr/ifftsize ; bin bandwidth
  kcnt init 0 ; counting vars
  krow init 0

  kOla[] init ifftsize ; overlap-add buffer
  kIn[] init ifftsize ; input buffer
  kSw[] init ifftsize
  kOut[][] init iolaps, ifftsize ; output buffers

  a1 diskin2 "fox.wav",1,0,1 ; audio input
```

```

ks expon 100, p3, 1000
asw vco2 k(a1), ks

/* every hopsize samples */
if kcnt == ihopsize then
  /* window and take FFT */
  kWin[] window kIn,krow*ihopsize
  kSpec[] rfft kWin
  kWin window kSw,krow*ihopsize
  kSpec2[] rfft kWin
  kProd[] cmplxprod kSpec, kSpec2

  /* IFFT + window */
  kRow[] rifft kProd
  kWin window kRow, krow*ihopsize
  /* place it on out buffer */
  kOut setrow kWin, krow

  /* zero the ola buffer */
  kOla = 0
  /* overlap-add */
  ki = 0
  until ki == iolaps do
    kRow getrow kOut, ki
    kOla = kOla + kRow
    ki += 1
  od

  /* update counters */
  krow = (krow+1)%iolaps
  kcnt = 0
endif

/* shift audio in/out of buffers */
kIn shiftin a1
kSw shiftin asw
a2 shiftout kOla
  out a2/iolaps

/* increment counter */
kcnt += ksmps

endin

</CsInstruments>

<CsScore>
i1 0 10
</CsScore>

</CsoundSynthesizer>

```

## See Also

Vectorial opcodes array opcodes

## Credits

Author: Victor Lazzarini  
 NUI Maynooth  
 2014

New in version 6.04

## getseed

getseed — Reads the global seed value.

### Description

Returns the global seed value used for all *x-class noise generators*.

### Syntax

```
ians getseed
```

```
kans getseed
```

### Performance

Reads the global seed value of the internal pseudo-random number generator.

### Credits

Author: John ffitch  
University of Bath/NUIM/Codemist Ltd.  
Bath, UK  
August 2015

New in Csound6.06

# gogobel

*gogobel* — Audio output is a tone related to the striking of a cow bell or similar.

## Description

Audio output is a tone related to the striking of a cow bell or similar. The method is a physical model developed from Perry Cook, but re-coded for Csound.

## Syntax

```
ares gogobel kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivfn
```

## Initialization

*ihrd* -- the hardness of the stick used in the strike. A range of 0 to 1 is used. 0.5 is a suitable value.

*ipos* -- where the block is hit, in the range 0 to 1.

*imp* -- a table of the strike impulses. The file *marmstk1.wav* [examples/marmstk1.wav] is a suitable function from measurements and can be loaded with a *GENO1* table. It is also available at <ftp://ftp.c-s.bath.ac.uk/pub/dream/documentation/sounds/modelling/>.

*ivfn* -- shape of vibrato, usually a sine table, created by a function.

## Performance

A note is played on a cowbell-like instrument, with the arguments as below.

*kamp* -- Amplitude of note.

*kfreq* -- Frequency of note played.

*kvibf* -- frequency of vibrato in Hertz. Suggested range is 0 to 12

*kvamp* -- amplitude of the vibrato

## Examples

Here is an example of the *gogobel* opcode. It uses the file *gogobel.csd* [examples/gogobel.csd], and *marmstk1.wav* [examples/marmstk1.wav],

### Example 386. Example of the *gogobel* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
-odac      ;;realtime audio out  
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
```

```
; For Non-realtime ouput leave only the line below:
; -o gogobel.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

    kfreq = 200
    ihrd = 0.5
    ipos = p4
    kvibf = 6.0
    kvamp = 0.3

    asig gogobel .9, kfreq, ihrd, ipos, 1, 6.0, 0.3, 2
        outs asig, asig
    endin
</CsInstruments>
<CsScore>
;audio file
f 1 0 256 1 "marmstkl.wav" 0 0 0
;sine wave for the vibrato
f 2 0 128 10 1

i 1 0.5 0.5 0.01
i 1 + 0.5 0.561
i 1 + 0.5 0.9
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

# goto

goto — Transfer control on every pass.

## Description

Transfer control to *label* on every pass. (Combination of *igoto* and *kgoto*)

## Syntax

```
goto label
```

where *label* is in the same instrument block and is not an expression.



### Note

Using *goto* not as part of an *if* statement (as in: goto end), will cause initialization to be skipped on all the code the *goto* jumps over. In performance, leaving some opcodes uninitialized will cause deletion of the note/event. In these cases, using *kgoto* (as in: kgoto end) might be preferred."

## Examples

Here is an example of the goto opcode. It uses the file *goto.csd* [examples/goto.csd].

### Example 387. Example of the goto opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      -iadc      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o goto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  a1 oscil 10000, 440, 1
  goto playit

  ; The goto will go to the playit label.
  ; It will skip any code in between like this comment.

playit:
  out a1
endin
```

```
</CsInstruments>
<CsScore>

; Table #1: a simple sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for one second.
i 1 0 1
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*cggoto, cigoto, ckgoto, if, igoto, kgoto, tigo, timeout*

## Credits

Example written by Kevin Conder.

Added a note by Jim Aikin.

# grain

grain — Generates granular synthesis textures.

## Description

Generates granular synthesis textures.

## Syntax

```
ares grain xamp, xpitch, xdens, kampoff, kpitchoff, kgdur, igfn, \  
      iwfn, imgdur [, igrnd]
```

## Initialization

*igfn* -- The ftable number of the grain waveform. This can be just a sine wave or a sampled sound.

*iwfn* -- Ftable number of the amplitude envelope used for the grains (see also *GEN20*).

*imgdur* -- Maximum grain duration in seconds. This is the biggest value to be assigned to *kgdur*.

*igrnd* (optional) -- if non-zero, turns off grain offset randomness. This means that all grains will begin reading from the beginning of the *igfn* table. If zero (the default), grains will start reading from random *igfn* table positions.

## Performance

*xamp* -- Amplitude of each grain.

*xpitch* -- Grain pitch. To use the original frequency of the input sound, use the formula:

$$\text{sndsr} / \text{ftlen}(\text{igfn})$$

where *sndsr* is the original sample rate of the *igfn* sound.

*xdens* -- Density of grains measured in grains per second. If this is constant then the output is synchronous granular synthesis, very similar to *fof*. If *xdens* has a random element (like added noise), then the result is more like asynchronous granular synthesis.

*kampoff* -- Maximum amplitude deviation from *xamp*. This means that the maximum amplitude a grain can have is *xamp* + *kampoff* and the minimum is *xamp*. If *kampoff* is set to zero then there is no random amplitude for each grain.

*kpitchoff* -- Maximum pitch deviation from *xpitch* in Hz. Similar to *kampoff*.

*kgdur* -- Grain duration in seconds. The maximum value for this should be declared in *imgdur*. If *kgdur* at any point becomes greater than *imgdur*, it will be truncated to *imgdur*.

The grain generator is based primarily on work and writings of Barry Truax and Curtis Roads.

## Examples

This example generates a texture with gradually shorter grains and wider amp and pitch spread. It uses the file *grain.csd* [examples/grain.csd], and *beats.wav* [examples/beats.wav].

### Example 388. Example of the grain opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o grain.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

insnd = 10
ibasfrq = 44100 / ftlen(insnd) ; Use original sample rate of insnd file

kamp   expseg .001, p3/2, .1, p3/2, .01 ; a swell in amplitude
kpitch line ibasfrq, p3, ibasfrq * .8
kdens  line 600, p3, 100
kaoff  line 0, p3, .1
kpoff  line 0, p3, ibasfrq * .5
kgdur  line .4, p3, .01
imaxgdur = .5

asigL  grain kamp, kpitch, kdens, kaoff, kpoff, kgdur, insnd, 5, imaxgdur, 0.0
asigR  grain kamp, kpitch, kdens, kaoff, kpoff, kgdur, insnd, 5, imaxgdur, 0.0
outs  asigL, asigR

endin
</CsInstruments>
<CsScore>
f5 0 512 20 2 ; Hanning window
f10 0 16384 1 "beats.wav" 0 0 0

i1 0 15
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Paris Smaragdis  
MIT  
May 1997

# grain2

grain2 — Easy-to-use granular synthesis texture generator.

## Description

Generate granular synthesis textures. *grain2* is simpler to use, but *grain3* offers more control.

## Syntax

```
ares grain2 kcps, kfmd, kgdur, iovrlp, kfn, iwfn [, irpow] \  
      [, iseed] [, imode]
```

## Initialization

*iovrlp* -- (fixed) number of overlapping grains.

*iwfn* -- function table containing window waveform (Use GEN20 to calculate *iwfn*).

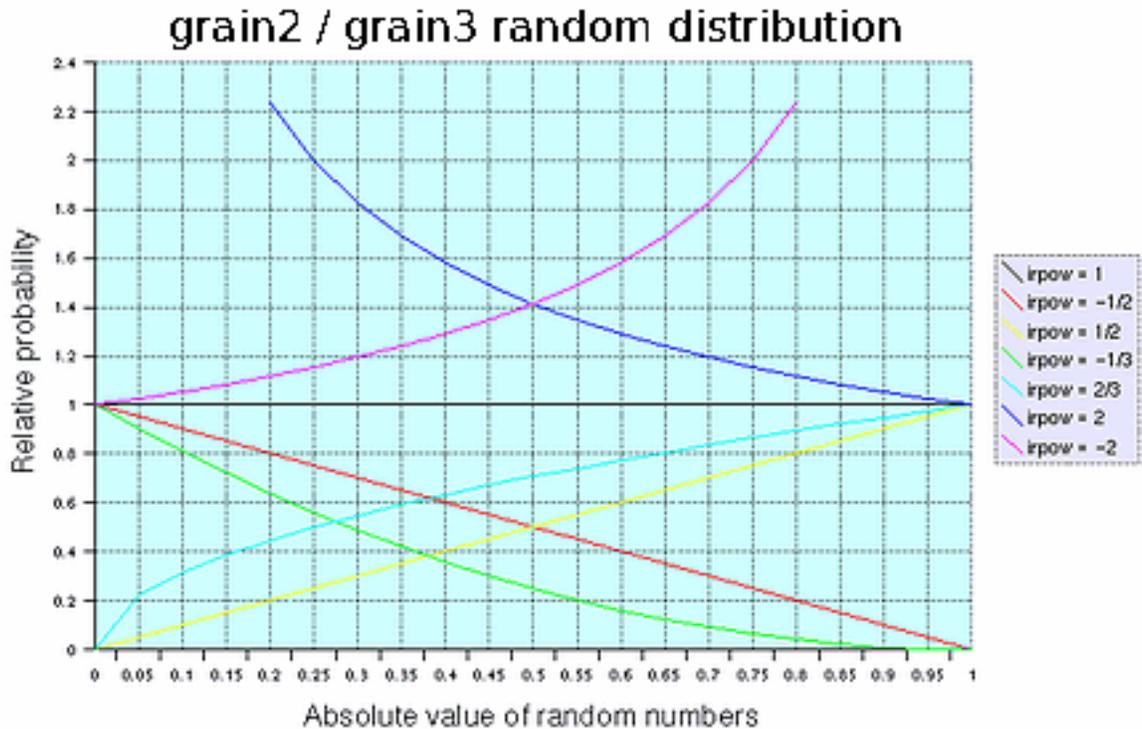
*irpow* (optional, default=0) -- this value controls the distribution of grain frequency variation. If *irpow* is positive, the random distribution ( $x$  is in the range -1 to 1) is

$\text{abs}(x) ^ ((1 / \text{irpow}) - 1)$ ;

for negative *irpow* values, it is

$(1 - \text{abs}(x)) ^ ((-1 / \text{irpow}) - 1)$

Setting *irpow* to -1, 0, or 1 will result in uniform distribution (this is also faster to calculate). The image below shows some examples for *irpow*. The default value of *irpow* is 0.



A graph of distributions for different values of *irpow*.

*iseed* (optional, default=0) -- seed value for random number generator (positive integer in the range 1 to 2147483646 ( $2^{31} - 2$ )). Zero or negative value seeds from current time (this is also the default).

*imode* (optional default=0) -- sum of the following values:

- 8: interpolate window waveform (slower).
- 4: do not interpolate grain waveform (fast, but lower quality).
- 2: grain frequency is continuously modified by *kcps* and *kfmd* (by default, each grain keeps the frequency it was launched with). This may be slower at high control rates.
- 1: skip initialization.

## Performance

*ares* -- output signal.

*kcps* -- grain frequency in Hz.

*kfmd* -- random variation (bipolar) in grain frequency in Hz.

*kgdur* -- grain duration in seconds. *kgdur* also controls the duration of already active grains (actually the speed at which the window function is read). This behavior does not depend on the *imode* flags.

*kfn* -- function table containing grain waveform. Table number can be changed at k-rate (this is useful to select from a set of band-limited tables generated by GEN30, to avoid aliasing).



## Note

*grain2* internally uses the same random number generator as *rnd31*. So reading *its documentation* is also recommended.

## Examples

Here is an example of the *grain2* opcode. It uses the file *grain2.csd* [examples/grain2.csd].

### Example 389. Example of the *grain2* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc   -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o grain2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 48000
kr = 750
ksmps = 64
nchnls = 2

/* square wave */
i_ ftgen 1, 0, 4096, 7, 1, 2048, 1, 0, -1, 2048, -1
/* window */
i_ ftgen 2, 0, 16384, 7, 0, 4096, 1, 4096, 0.3333, 8192, 0
/* sine wave */
i_ ftgen 3, 0, 1024, 10, 1
/* room parameters */
i_ ftgen 7, 0, 64, -2, 4, 50, -1, -1, -1, 11, \
    1, 26.833, 0.05, 0.85, 10000, 0.8, 0.5, 2, \
    1, 1.753, 0.05, 0.85, 5000, 0.8, 0.5, 2, \
    1, 39.451, 0.05, 0.85, 7000, 0.8, 0.5, 2, \
    1, 33.503, 0.05, 0.85, 7000, 0.8, 0.5, 2, \
    1, 36.151, 0.05, 0.85, 7000, 0.8, 0.5, 2, \
    1, 29.633, 0.05, 0.85, 7000, 0.8, 0.5, 2

ga01 init 0

/* generate bandlimited square waves */

i0 = 0
loop1:
imaxh = sr / (2 * 440.0 * exp(log(2.0) * (i0 - 69) / 12))
i_ ftgen i0 + 256, 0, 4096, -30, 1, 1, imaxh
i0 = i0 + 1
if (i0 < 127.5) igoto loop1

instr 1

p3 = p3 + 0.2

/* note velocity */
iamp = 0.0039 + p5 * p5 / 16192
/* vibrato */
kcps oscili 1, 8, 3
```

```

kenv linseg 0, 0.05, 0, 0.1, 1, 1, 1
/* frequency */
kcps = (kcps * kenv * 0.01 + 1) * 440 * exp(log(2) * (p4 - 69) / 12)
/* grain ftable */
kfn = int(256 + 69 + 0.5 + 12 * log(kcps / 440) / log(2))
/* grain duration */
kgdur port 100, 0.1, 20
kgdur = kgdur / kcps

a1 grain2 kcps, kcps * 0.02, kgdur, 50, kfn, 2, -0.5, 22, 2
a1 butterlp a1, 3000
a2 grain2 kcps, kcps * 0.02, 4 / kcps, 50, kfn, 2, -0.5, 23, 2
a2 butterbp a2, 12000, 8000
a2 butterbp a2, 12000, 8000
aenv1 linseg 0, 0.01, 1, 1, 1
aenv2 linseg 3, 0.05, 1, 1, 1
aenv3 linseg 1, p3 - 0.2, 1, 0.07, 0, 1, 0

a1 = aenv1 * aenv3 * (a1 + a2 * 0.7 * aenv2)

ga01 = ga01 + a1 * 10000 * iamp

    endin

/* output instr */

    instr 81

i1 = 0.000001
aLl, aLh, aRl, aRh spat3di ga01 + i1*i1*i1*i1, 3.0, 4.0, 0.0, 0.5, 7, 4
ga01 = 0
aLl butterlp aLl, 800.0
aRl butterlp aRl, 800.0

    outs aLl + aLh, aRl + aRh

    endin

</CsInstruments>
<CsScore>

t 0 60

i 1 0.0 1.3 60 127
i 1 2.0 1.3 67 127
i 1 4.0 1.3 64 112
i 1 4.0 1.3 72 112

i 81 0 6.4

e

</CsScore>
</CsoundSynthesizer>

```

## See Also

*grain3*

## Credits

Author: Istvan Varga

New in version 4.15

Updated April 2002 by Istvan Varga

# grain3

`grain3` — Generate granular synthesis textures with more user control.

## Description

Generate granular synthesis textures. *grain2* is simpler to use but *grain3* offers more control.

## Syntax

```
ares grain3 kcps, kphs, kfmd, kpmd, kgdur, kdens, imaxovr, kfn, iwfn, \  
      kfrpow, krpow [, iseed] [, imode]
```

## Initialization

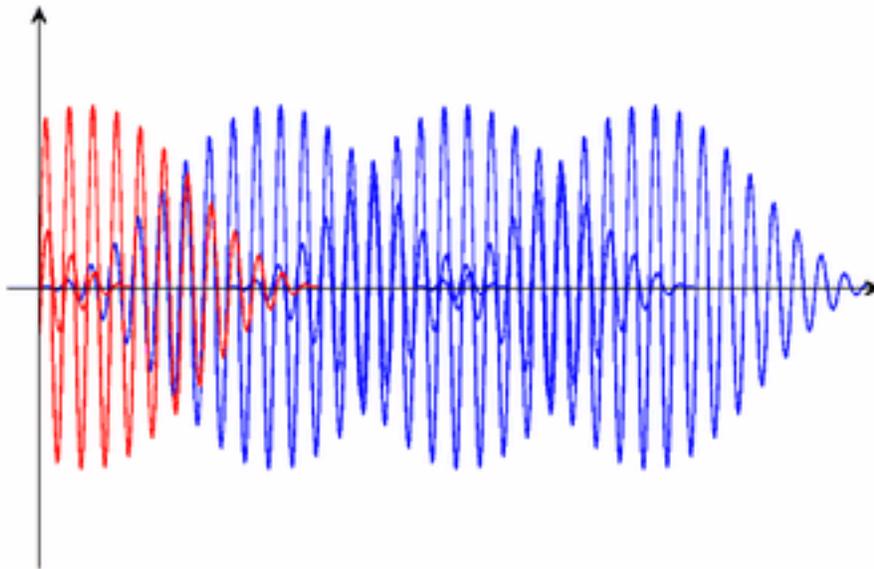
*imaxovr* -- maximum number of overlapping grains. The number of overlaps can be calculated by (*kdens* \* *kgdur*); however, it can be overestimated at no cost in rendering time, and a single overlap uses (depending on system) 16 to 32 bytes of memory.

*iwfn* -- function table containing window waveform (Use GEN20 to calculate *iwfn*).

*iseed* (optional, default=0) -- seed value for random number generator (positive integer in the range 1 to 2147483646 ( $2^{31} - 2$ )). Zero or negative value seeds from current time (this is also the default).

*imode* (optional, default=0) -- sum of the following values:

- 64: synchronize start phase of grains to *kcps*.
- 32: start all grains at integer sample location. This may be faster in some cases, however it also makes the timing of grain envelopes less accurate.
- 16: do not render grains with start time less than zero. (see the image below; this option turns off grains marked with red on the image).
- 8: interpolate window waveform (slower).
- 4: do not interpolate grain waveform (fast, but lower quality).
- 2: grain frequency is continuously modified by *kcps* and *kfmd* (by default, each grain keeps the frequency it was launched with). This may be slower at high control rates. It also controls phase modulation (*kphs*).
- 1: skip initialization.



A diagram showing grains with a start time less than zero in red.

## Performance

*ares* -- output signal.

*kcps* -- grain frequency in Hz.

*kphs* -- grain phase. This is the location in the grain waveform table, expressed as a fraction (between 0 to 1) of the table length.

*kfmd* -- random variation (bipolar) in grain frequency in Hz.

*kpmf* -- random variation (bipolar) in start phase.

*kgdur* -- grain duration in seconds. *kgdur* also controls the duration of already active grains (actually the speed at which the window function is read). This behavior does not depend on the *imode* flags.

*kdens* -- number of grains per second.

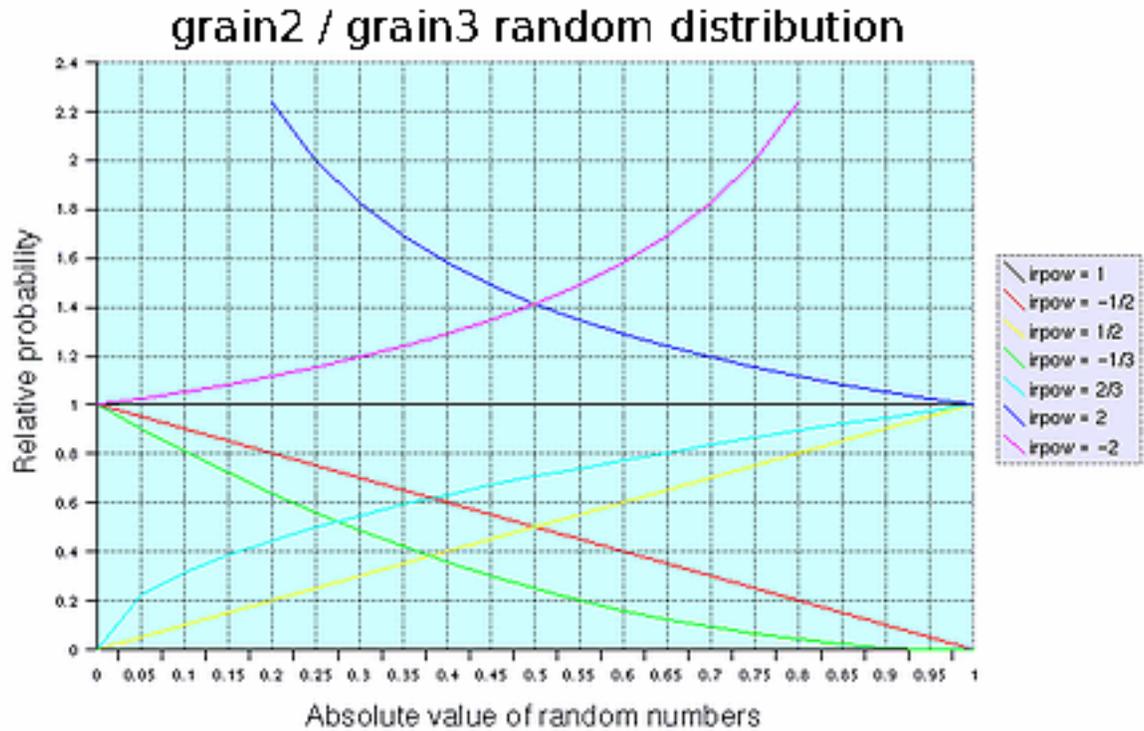
*kfrpow* -- this value controls the distribution of grain frequency variation. If *kfrpow* is positive, the random distribution ( $x$  is in the range -1 to 1) is

$$\text{abs}(x)^{\left(\frac{1}{kfrpow} - 1\right)}$$

; for negative *kfrpow* values, it is

$$\left(1 - \text{abs}(x)\right)^{\left(\frac{-1}{kfrpow} - 1\right)}$$

Setting *kfrpow* to -1, 0, or 1 will result in uniform distribution (this is also faster to calculate). The image below shows some examples for *kfrpow*. The default value of *kfrpow* is 0.



A graph of distributions for different values of *kfrpow*.

*kprpow* -- distribution of random phase variation (see *kfrpow*). Setting *kphs* and *kpmid* to 0.5, and *kprpow* to 0 will emulate *grain2*.

*kfn* -- function table containing grain waveform. Table number can be changed at k-rate (this is useful to select from a set of band-limited tables generated by GEN30, to avoid aliasing).



### Note

*grain3* internally uses the same random number generator as *rnd31*. So reading *its documentation* is also recommended.

## Examples

Here is an example of the *grain3* opcode. It uses the file *grain3.csd* [examples/grain3.csd].

### Example 390. Example of the *grain3* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o grain3.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```

sr = 48000
kr = 1000
ksmps = 48
nchnls = 1

/* Bartlett window */
itmp ftgen 1, 0, 16384, 20, 3, 1
/* sawtooth wave */
itmp ftgen 2, 0, 16384, 7, 1, 16384, -1
/* sine */
itmp ftgen 4, 0, 1024, 10, 1
/* window for "soft sync" with 1/32 overlap */
itmp ftgen 5, 0, 16384, 7, 0, 256, 1, 7936, 1, 256, 0, 7936, 0
/* generate bandlimited sawtooth waves */
itmp ftgen 3, 0, 4096, -30, 2, 1, 2048
icnt = 0
loop01:
; 100 tables for 8 octaves from 30 Hz
ifrq = 30 * exp(log(2) * 8 * icnt / 100)
itmp ftgen icnt + 100, 0, 4096, -30, 3, 1, sr / (2 * ifrq)
icnt = icnt + 1
if (icnt < 99.5) igoto loop01
/* convert frequency to table number */
#define FRQ2FNUM(xout'xcps'xbsfn) #

$xout = int((xbsfn) + 0.5 + (100 / 8) * log((xcps) / 30) / log(2))
$xout limit $xout, $xbsfn, $xbsfn + 99

#

/* instr 1: pulse width modulated grains */

instr 1

kfrq = 523.25 ; frequency
$FRQ2FNUM(kfnun'kfrq'100) ; table number
kfmd = kfrq * 0.02 ; random variation in frequency
kgdur = 0.2 ; grain duration
kdens = 200 ; density
iseed = 1 ; random seed

kphs oscili 0.45, 1, 4 ; phase

a1 grain3 kfrq, 0, kfmd, 0.5, kgdur, kdens, 100, \
kfnun, 1, -0.5, 0, iseed, 2
a2 grain3 kfrq, 0.5 + kphs, kfmd, 0.5, kgdur, kdens, 100, \
kfnun, 1, -0.5, 0, iseed, 2

; de-click
aenv linseg 0, 0.01, 1, p3 - 0.05, 1, 0.04, 0, 1, 0

out aenv * 2250 * (a1 - a2)

endin

/* instr 2: phase variation */

instr 2

kfrq = 220 ; frequency
$FRQ2FNUM(kfnun'kfrq'100) ; table number
kgdur = 0.2 ; grain duration
kdens = 200 ; density
iseed = 2 ; random seed

kprdst expon 0.5, p3, 0.02 ; distribution

```

```
a1 grain3 kfrq, 0.5, 0, 0.5, kgdur, kdens, 100, \  
  kfnum, 1, 0, -kprdst, iseed, 64  
  
; de-click  
aenv linseg 0, 0.01, 1, p3 - 0.05, 1, 0.04, 0, 1, 0  
  
  out aenv * 1500 * a1  
  
  endin  
  
/* instr 3: "soft sync" */  
  
  instr 3  
  
  kdens = 130.8 ; base frequency  
  kgdur = 2 / kdens ; grain duration  
  
  kfrq expon 880, p3, 220 ; oscillator frequency  
  $FRQ2FNUM(kfnum'kfrq'100) ; table number  
  
a1 grain3 kfrq, 0, 0, 0, kgdur, kdens, 3, kfnum, 5, 0, 0, 0, 2  
a2 grain3 kfrq, 0.667, 0, 0, kgdur, kdens, 3, kfnum, 5, 0, 0, 0, 2  
  
; de-click  
aenv linseg 0, 0.01, 1, p3 - 0.05, 1, 0.04, 0, 1, 0  
  
  out aenv * 10000 * (a1 - a2)  
  
  endin  
  
</CsInstruments>  
<CsScore>  
  
  t 0 60  
  i 1 0 3  
  i 2 4 3  
  i 3 8 3  
  e  
  
</CsScore>  
</CsSoundSynthesizer>
```

## See Also

*grain2*

## Credits

Author: Istvan Varga

New in version 4.15

Updated April 2002 by Istvan Varga

# granule

*granule* — A more complex granular synthesis texture generator.

## Description

The *granule* unit generator is more complex than *grain*, but does add new possibilities.

*granule* is a Csound unit generator which employs a wavetable as input to produce granularly synthesized audio output. Wavetable data may be generated by any of the GEN subroutines such as *GEN01* which reads an audio data file into a wavetable. This enable a sampled sound to be used as the source for the grains. Up to 128 voices are implemented internally. The maximum number of voices can be increased by redefining the variable MAXVOICE in the grain4.h file. *granule* has a build-in random number generator to handle all the random offset parameters. Thresholding is also implemented to scan the source function table at initialization stage. This facilitates features such as skipping silence passage between sentences.

The characteristics of the synthesis are controlled by 22 parameters. *xamp* is the amplitude of the output and it can be either audio rate or control rate variable.

## Syntax

```
ares granule xamp, ivoice, iratio, imode, ithd, ifn, ipshift, igskip, \  
    igskip_os, ilength, kgap, igap_os, kgsz, igsz_os, iatt, idec \  
    [, iseed] [, ipitch1] [, ipitch2] [, ipitch3] [, ipitch4] [, ifnenv]
```

## Initialization

*ivoice* -- number of voices.

*iratio* -- ratio of the speed of the gskip pointer relative to output audio sample rate. eg. 0.5 will be half speed.

*imode* -- +1 grain pointer move forward (same direction of the gskip pointer), -1 backward (oppose direction to the gskip pointer) or 0 for random.

*ithd* -- threshold, if the sampled signal in the wavetable is smaller then *ithd*, it will be skipped.

*ifn* -- function table number of sound source.

*ipshift* -- pitch shift control. If *ipshift* is 0, pitch will be set randomly up and down an octave. If *ipshift* is 1, 2, 3 or 4, up to four different pitches can be set amount the number of voices defined in *ivoice*. The optional parameters *ipitch1*, *ipitch2*, *ipitch3* and *ipitch4* are used to quantify the pitch shifts.

*igskip* -- initial skip from the beginning of the function table in sec.

*igskip\_os* -- gskip pointer random offset in sec, 0 will be no offset.

*ilength* -- length of the table to be used starting from *igskip* in sec.

*igap\_os* -- gap random offset in % of the gap size, 0 gives no offset.

*igsz\_os* -- grain size random offset in % of grain size, 0 gives no offset.

*iatt* -- attack of the grain envelope in % of grain size.

*idec* -- decade of the grain envelope in % of grain size.

*iseed* (optional, default=0.5) -- seed for the random number generator.

*ipitch1*, *ipitch2*, *ipitch3*, *ipitch4* (optional, default=1) -- pitch shift parameter, used when *ipshift* is set to 1, 2, 3 or 4. Time scaling technique is used in pitch shift with linear interpolation between data points. Default value is 1, the original pitch.

*ifnenv* (optional, default=0) -- function table number to be used to generate the shape of the envelope.

## Performance

*xamp* -- amplitude.

*kgap* -- gap between grains in sec.

*kgsiz*e -- grain size in sec.

## Examples

Here is an example of the granule opcode. It uses the file *granule.csd* [examples/granule.csd], and *mary.wav* [examples/mary.wav].

### Example 391. Example of the granule opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o granule.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 2
instr 1
;
k1      linseg 0,0.5,1,(p3-p2-1),1,0.5,0
a1      granule p4*k1,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14,p15,\
p16,p17,p18,p19,p20,p21,p22,p23,p24
a2      granule p4*k1,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14,p15,\
p16,p17,p18,p19, p20+0.17,p21,p22,p23,p24
outs a1,a2
endin

</CsInstruments>
<CsScore>

; f statement read sound file mary.wav in the SFDIR
; directory into f-table 1
f1      0 262144 1 "mary.wav" 0 0 0
i1      0 10 2000 64 0.5 0 0 1 4 0 0.005 5 0.01 50 0.02 50 30 30 0.39 \
```

```
1 1.42 0.29 2
```

```
e
```

```
</CsScore>
</CsoundSynthesizer>
```

The above example reads a sound file called *mary.wav* into wavetable number 1 with 262,144 samples. It generates 10 seconds of stereo audio output using the wavetable. In the orchestra file, all parameters required to control the synthesis are passed from the score file. A *linseg* function generator is used to generate an envelope with 0.5 second of linear attack and decay. Stereo effect is generated by using different seeds for the two *granule* function calls. In the example, 0.17 is added to p20 before passing into the second *granule* call to ensure that all of the random offset events are different from the first one.

In the score file, the parameters are interpreted as:

Parameter	Interpreted As
p5 ( <i>ivoice</i> )	the number of voices is set to 64
p6 ( <i>iratio</i> )	set to 0.5, it scans the wavetable at half of the speed of the audio output rate
p7 ( <i>imode</i> )	set to 0, the grain pointer only move forward
p8 ( <i>ithd</i> )	set to 0, skipping the thresholding process
p9 ( <i>ifn</i> )	set to 1, function table number 1 is used
p10 ( <i>ipshift</i> )	set to 4, four different pitches are going to be generated
p11 ( <i>igskip</i> )	set to 0 and p12 ( <i>igskip_os</i> ) is set to 0.005, no skipping into the wavetable and a 5 mSec random offset is used
p13 ( <i>ilength</i> )	set to 5, 5 seconds of the wavetable is to be used
p14 ( <i>kgap</i> )	set to 0.01 and p15 ( <i>igap_os</i> ) is set to 50, 10 mSec gap with 50% random offset is to be used
p16 ( <i>kgsiz</i> )	set to 0.02 and p17 ( <i>igsize_os</i> ) is set to 50, 20 mSec grain with 50% random offset is used
p18 ( <i>iatt</i> ) and p19 ( <i>idec</i> )	set to 30, 30% of linear attack and decade is applied to the grain
p20 ( <i>iseed</i> )	seed for the random number generator is set to 0.39
p21 - p24	pitches set to 1 which is the original pitch, 1.42 which is a 5th up, 0.29 which is a 7th down and finally 2 which is an octave up.

## Credits

Author: Allan Lee

Belfast

1996

New in version 3.35

# guiro

guiro — Semi-physical model of a guiro sound.

## Description

*guiro* is a semi-physical model of a guiro sound. It is one of the PhISEM percussion opcodes. PhISEM (Physically Informed Stochastic Event Modeling) is an algorithmic approach for simulating collisions of multiple independent sound producing objects.

## Syntax

```
ares guiro kamp, idettack [, inum] [, idamp] [, imaxshake] [, ifreq] [, ifreq1]
```

## Initialization

*idettack* -- period of time over which all sound is stopped

*inum* (optional) -- The number of beads, teeth, bells, timbrels, etc. If zero, the default value is 128.

*idamp* (optional) -- the damping factor of the instrument. *Not used.*

*imaxshake* (optional, default=0) -- amount of energy to add back into the system. The value should be in range 0 to 1.

*ifreq* (optional) -- the main resonant frequency. The default value is 2500 Hz.

*ifreq1* (optional) -- the first resonant frequency. Default value here is 4000 Hz

## Performance

*kamp* -- Amplitude of output. Note: As these instruments are stochastic, this is only an approximation.

## Examples

Here is an example of the guiro opcode. It uses the file *guiro.csd* [examples/guiro.csd].

### Example 392. Example of the guiro opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsoundOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o guiro.wav -W ;; for file output any platform
</CsoundOptions>
<CsoundInstruments>

sr = 44100
```

```
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

a1 guiro .8, p4
   outs a1, a1

endin
</CsInstruments>
<CsScore>

i1 0 1 1
i1 + 1 .01
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*bamboo, dripwater, sleighbells, tambourine*

## Credits

Author: Perry Cook, part of the PhISEM (Physically Informed Stochastic Event Modeling)  
Adapted by John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 4.07

Added notes by Rasmus Ekman on May 2002.

# harmon

harmon — Analyze an audio input and generate harmonizing voices in synchrony.

## Description

Analyze an audio input and generate harmonizing voices in synchrony.

## Syntax

```
ares harmon asig, kestfrq, kmaxvar, kgenfreq1, kgenfreq2, imode, \  
    iminfrq, iprd
```

## Initialization

*imode* -- interpreting mode for the generating frequency inputs *kgenfreq1*, *kgenfreq2*. 0: input values are ratios with respect to the audio signal analyzed frequency. 1: input values are the actual requested frequencies in Hz.

*iminfrq* -- the lowest expected frequency (in Hz) of the audio input. This parameter determines how much of the input is saved for the running analysis, and sets a lower bound on the internal pitch tracker.

*iprd* -- period of analysis (in seconds). Since the internal pitch analysis can be time-consuming, the input is typically analyzed only each 20 to 50 milliseconds.

## Performance

*kestfrq* -- estimated frequency of the input.

*kmaxvar* -- the maximum variance (expects a value between 0 and 1).

*kgenfreq1* -- the first generated frequency.

*kgenfreq2* -- the second generated frequency.

This unit is a harmonizer, able to provide up to two additional voices with the same amplitude and spectrum as the input. The input analysis is assisted by two things: an input estimated frequency *kestfrq* (in Hz), and a fractional maximum variance *kmaxvar* about that estimate which serves to limit the size of the search. Once the real input frequency is determined, the most recent pulse shape is used to generate the other voices at their requested frequencies.

The three frequency inputs can be derived in various ways from a score file or MIDI source. The first is the expected pitch, with a variance parameter allowing for inflections or inaccuracies; if the expected pitch is zero the harmonizer will be silent. The second and third pitches control the output frequencies; if either is zero the harmonizer will output only the non-zero request; if both are zero the harmonizer will be silent. When the requested frequency is higher than the input, the process requires additional computation due to overlapped output pulses. This is currently limited for efficiency reasons, with the result that only one voice can be higher than the input at any one time.

This unit is useful for supplying a background chorus effect on demand, or for correcting the pitch of a faulty input vocal. There is essentially no delay between input and output. Output includes only the generated parts, and does not include the input.

## Examples

Here is an example of the harmon opcode. It uses the file *harmon.csd* [examples/harmon.csd].

### Example 393. Example of the harmon opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o harmon.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

aout diskin2 "sing.wav", 1

kestfrq = p4 ;different estimated frequency
kmaxvar = 0.1
imode = 1
iminfrq = 100
iprd = 0.02

asig harmon aout, kestfrq, kmaxvar, kestfrq*.5, kestfrq*4, \
      imode, iminfrq, iprd
      outs (asig + aout)*.6, (asig + aout)*.6 ;mix dry&wet signal

endin
</CsInstruments>
<CsScore>

i 1 0 2.7 100
i 1 + . 200
i 1 + . 500
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Barry L. Vercoe  
M.I.T., Cambridge, Mass  
1997

New in version 3.47

# harmon2

**harmon2** — Analyze an audio input and generate harmonizing voices in synchrony with formants preserved.

## Description

Generate harmonizing voices with formants preserved.

## Syntax

```
ares harmon2 asig, koct, kfrq1, kfrq2, icpsmode, ilowest[, ipolarity]

ares harmon3 asig, koct, kfrq1, \
    kfrq2, kfrq3, icpsmode, ilowest[, ipolarity]

ares harmon4 asig, koct, kfrq1, \
    kfrq2, kfrq3, kfrq4, icpsmode, ilowest[, ipolarity]
```

## Initialization

*icpsmode* -- interpreting mode for the generating frequency inputs *kfrq1*, *kfrq2*, *kfrq3* and *kfrq4*: 0: input values are ratios w.r.t. the cps equivalent of *koct*. 1: input values are the actual requested frequencies in cps.

*ilowest* -- lowest value of the *koct* input for which harmonizing voices will be generated.

*ipolarity* -- polarity of *asig* input, 1 = positive glottal pulses, 0 = negative. Default is 1.

## Performance

**Harmon2**, **harmon3** and **harmon4** are high-performance harmonizers, able to provide up to four pitch-shifted copies of the input *asig* with spectral formants preserved. The pitch-shifting algorithm requires an accurate running estimate (*koct*, in decimal oct units) of the pitched content of *asig*, normally gained from an independent pitch tracker such as *specptrk*. The algorithm then isolates the most recent full pulse within *asig*, and uses this to generate the other voices at their required pulse rates.

If the frequency (or ratio) presented to *kfrq1*, *kfrq2*, *kfrq3* or *kfrq4* is zero, then no signal is generated for that voice. If any of them is non-zero, but the *koct* input is below the value *ilowest*, then that voice will output a direct copy of the input *asig*. As a consequence, the data arriving at the k-rate inputs can variously cause the generated voices to be turned on or off, to pass a direct copy of a non-voiced fricative source, or to harmonize the source according to some constructed algorithm. The transition from one mode to another is cross-faded, giving seamless alternating between voiced (harmonized) and non-voiced fricatives during spoken or sung input.

*harmon2*, *harmon3*, *harmon4* are especially matched to the output of *specptrk*. The latter generates pitch data in decimal octave format; it also emits its base value if no pitch is identified (as in fricative noise) and emits zero if the energy falls below a threshold, so that *harmon2*, *harmon3*, *harmon4* can be set to pass the direct signal in both cases. Of course, any other form of pitch estimation could also be used. Since pitch trackers usually incur a slight delay for accurate estimation (for *specptrk* the delay is printed by the *spectrum* unit), it is normal to delay the audio signal by the same amount so that *harmon2*, *harmon3*, *harmon4* can work from a fully concurrent estimate.

## Examples

Here is an example of the `harmon2` opcode. It uses the file `harmon.csd` [examples/harmon.csd].

### Example 394. Example of the `harmon2` opcode.

```
a1,a2      ins                               ; get mic input
w1         spectrum a1, .02, 7, 24, 12, 1, 3   ; and examine it
koct,kamp  specptrk w1, 1, 6.5, 9.5, 7.5, 10, 7, .7, 0, 3, 1
a3         delay a1, .065                     ; allow for ptrk delay
a4         harmon2 a3, koct, 1.25, 0.75, 0, 6.9 ; output a fixed 6-4 harmony
          outs a3, a4                         ; as well as the original
```

Here is a complete example of the `harmon3` opcode. It uses the file `harmon3.csd` [examples/harmon3.csd].

### Example 395. Example of the `harmon3` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o harmon3.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ilow = p4      ;lowest value to harmonize
aout diskin2 "sing.wav", 1, 0, 1
koct, kamp pitch aout, .01, 6, 10, 10 ;track pitch
asig harmon3 aout, koct, .9, 1.5, 0.7, 0, ilow
      outs (asig + aout)*.4, (asig + aout)*.4 ;mix dry&wet signal

endin
</CsInstruments>
<CsScore>

i1 0 2.2 8.8
i1 3 2.2 8.2
i1 6 2.2 7.0

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Barry L. Vercoe  
M.I.T., Cambridge, Mass  
2006

New in version 5.04

# hdf5read

hdf5read — Read signals and arrays from an hdf5 file.

## Description

*hdf5read* reads *N* signals and arrays from a specified hdf5 file.

## Syntax

```
xout1[, xout2, xout3, ..., xoutN] hdf5read ifilename, ivariablename1[, ivariablename2, ivariablename3,
```

## Initialization

*ifilename* -- the hdf5 file's name (in double-quotes).

*ivariablename1*[, *ivariablename2*, *ivariablename3*, ..., *ivariablenameN*] -- the names of the datasets (in double-quotes) to be read from the hdf5 file, if the dataset name is suffixed with an asterisk, e.g. "my-dataset\*", the entire dataset is copied to the array regardless of array type.

## Performance

*xout1*,... *xoutN* -- The specified types of variables that the hdf5 datasets are to be read as. Datasets with a rank larger than 1 must be read as arrays, i-rate signals must also be read as i-rate signals. Other than these restrictions datasets may be read as any type of array or signal. When reading has reached the end of a dataset it no longer outputs any new values.

## Examples

Here is a simple example of the hdf5read opcode. It uses the file *hdf5read.csd* [examples/hdf5read.csd].

### Example 396. Example of the hdf5read opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
nchnls = 2
0dbfs = 1
ksmps = 8
sr = 44100

instr hdf5read

  aArray[], aVar, kVar hdf5read "example.h5", "aArray", "aVar", "kVar" ; Open hdf5 file and read vari

  aLeft = (aArray[0][0] + aArray[0][1] + aVar) / 3 ; Add audio signals together for stereo out
  aRight = (aArray[1][0] + aArray[1][1] + aVar) / 3
```

```
    outs aLeft * kVar, aRight * kVar ; Multiply audio signals by k-rate signal
endin

</CsInstruments>
<CsScore>

i "hdf5read" 0 1

</CsScore>
</CsoundSynthesizer>
```

## See Also

*hdf5write*

## Credits

Author: Edward Costello;  
NUIM, 2014

# hdf5write

hdf5write — Write signals and arrays to an hdf5 file.

## Description

*hdf5write* writes *N* signals and arrays to a specified hdf5 file.

## Syntax

```
hdf5write ifilename, xout1[, xout2, xout3, ..., xoutN]
```

## Initialization

*ifilename* -- the hdf5 file's name (in double-quotes). If the file does not exist it will be created.

## Performance

*xout1,...* *xoutN* -- signals or arrays to be written to the hdf5 file. This opcode accepts i-rate, k-rate, a-rate signals or i-rate, k-rate, a-rate arrays of any dimension. These signals or arrays are written to a dataset within the hdf5 file using the same variable name as in Csound. For example, if the Csound variable is called 'ksignal', then the name of the hdf5 dataset is 'ksignal'. Any number and multiple types of datasets may be written at a time.

## Examples

Here is a simple example of the hdf5write opcode. It uses the file *hdf5write.csd* [examples/hdf5write.csd].

### Example 397. Example of the hdf5write opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
nchnls = 2
Odbfs = 1
ksmps = 8
sr = 44100

instr hdf5write

  aArray[] init 2,2 ; Initialise a 2 X 2 a-rate array

  aArray[0][0] vco2 0.2, 100 ; Fill array with vco2 signals
  aArray[0][1] vco2 0.4, 200
  aArray[1][0] vco2 0.8, 300
  aArray[1][1] vco2 1, 400

  aVar vco2 0.2, 100 ; Initialise an a-rate variable with a vco2 signal
```

```
kVar phasor 1 ; Initialise a k-rate variable with a phasor signal

hdf5write "example.h5", aArray, aVar, kVar ; Write variables to an hdf5 file

endin

</CsInstruments>
<CsScore>

i "hdf5write" 0 1

</CsScore>
</CsoundSynthesizer>
```

## See Also

*hdf5read*

## Credits

Author: Edward Costello;  
NUIM, 2014

# hilbert

hilbert — A Hilbert transformer.

## Description

An IIR implementation of a Hilbert transformer.

## Syntax

```
ar1, ar2 hilbert asig
```

## Performance

*asig* -- input signal

*ar1* -- sine output of *asig*

*ar2* -- cosine output of *asig*

*hilbert* is an IIR filter based implementation of a broad-band 90 degree phase difference network. The input to *hilbert* is an audio signal, with a frequency range from 15 Hz to 15 kHz. The outputs of *hilbert* have an identical frequency response to the input (i.e. they sound the same), but the two outputs have a constant phase difference of 90 degrees, plus or minus some small amount of error, throughout the entire frequency range. The outputs are in quadrature.

*hilbert* is useful in the implementation of many digital signal processing techniques that require a signal in phase quadrature. *ar1* corresponds to the cosine output of *hilbert*, while *ar2* corresponds to the sine output. The two outputs have a constant phase difference throughout the audio range that corresponds to the phase relationship between cosine and sine waves.

Internally, *hilbert* is based on two parallel 6th-order allpass filters. Each allpass filter implements a phase lag that increases with frequency; the difference between the phase lags of the parallel allpass filters at any given point is approximately 90 degrees.

Unlike an FIR-based Hilbert transformer, the output of *hilbert* does not have a linear phase response. However, the IIR structure used in *hilbert* is far more efficient to compute, and the nonlinear phase response can be used in the creation of interesting audio effects, as in the second example below.

## Examples

The first example implements frequency shifting, or single sideband amplitude modulation. Frequency shifting is similar to ring modulation, except the upper and lower sidebands are separated into individual outputs. By using only one of the outputs, the input signal can be "detuned," where the harmonic components of the signal are shifted out of harmonic alignment with each other, e.g. a signal with harmonics at 100, 200, 300, 400 and 500 Hz, shifted up by 50 Hz, will have harmonics at 150, 250, 350, 450, and 550 Hz.

Here is the first example of the *hilbert* opcode. It uses the file *hilbert.csd* [examples/hilbert.csd], and *beats.wav* [examples/beats.wav].

**Example 398. Example of the hilbert opcode implementing frequency shifting.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o hilbert.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
  idur = p3
  ; Initial amount of frequency shift.
  ; It can be positive or negative.
  ibegshift = p4
  ; Final amount of frequency shift.
  ; It can be positive or negative.
  iendshift = p5

  ; A simple envelope for determining the
  ; amount of frequency shift.
  kfreq linseg ibegshift, idur, iendshift

  ; Use the sound of your choice.
  ain diskin2 "beats.wav", 1, 0, 1

  ; Phase quadrature output derived from input signal.
  areal, aimag hilbert ain

  ; Quadrature oscillator.
  asin oscili 1, kfreq, 1
  acos oscili 1, kfreq, 1, .25

  ; Use a trigonometric identity.
  ; See the references for further details.
  amod1 = areal * acos
  amod2 = aimag * asin

  ; Both sum and difference frequencies can be
  ; output at once.
  ; aupshift corresponds to the sum frequencies.
  aupshift = (amod1 - amod2) * 0.7
  ; adownshift corresponds to the difference frequencies.
  adownshift = (amod1 + amod2) * 0.7

  ; Notice that the adding of the two together is
  ; identical to the output of ring modulation.

  outs aupshift, aupshift
endin

</CsInstruments>
<CsScore>

; Sine table for quadrature oscillator.

```

```
f 1 0 16384 10 1

; Starting with no shift, ending with all
; frequencies shifted up by 2000 Hz.
i 1 0 6 0 2000

; Starting with no shift, ending with all
; frequencies shifted down by 250 Hz.
i 1 7 6 0 -250
e
```

```
</CsScore>
</CsoundSynthesizer>
```

The second example is a variation of the first, but with the output being fed back into the input. With very small shift amounts (i.e. between 0 and +-6 Hz), the result is a sound that has been described as a “barberpole phaser” or “Shepard tone phase shifter.” Several notches appear in the spectrum, and are constantly swept in the direction opposite that of the shift, producing a filtering effect that is reminiscent of Risset’s “endless glissando”.

Here is the second example of the hilbert opcode. It uses the file *hilbert\_barberpole.csd* [examples/hilbert\_barberpole.csd].

### Example 399. Example of the hilbert opcode sounding like a “barberpole phaser”.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc      -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o hilbert_barberpole.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
; kr must equal sr for the barberpole effect to work.
kr = 44100
ksmps = 1
nchnls = 2

; Instrument #1
instr 1
  idur = p3
  ibegshift = p4
  iendshift = p5

  ; sawtooth wave, not bandlimited
  asaw phasor 100
  ; add offset to center phasor amplitude between -.5 and .5
  asaw = asaw - .5
  ; sawtooth wave, with amplitude of 10000
  ain = asaw * 20000

  ; The envelope of the frequency shift.
  kfreq linseg ibegshift, idur, iendshift

  ; Phase quadrature output derived from input signal.
```

```
areal, aimag hilbert ain

; The quadrature oscillator.
asin oscili 1, kfreq, 1
acos oscili 1, kfreq, 1, .25

; Based on trigonometric identities.
amod1 = areal * acos
amod2 = aimag * asin

; Calculate the up-shift and down-shift.
aupshift = (amod1 + amod2) * 0.7
adownshift = (amod1 - amod2) * 0.7

; Mix in the original signal to achieve the barberpole effect.
amix1 = aupshift + ain
amix2 = adownshift + ain

; Make sure the output doesn't get louder than the original signal.
aout1 balance amix1, ain
aout2 balance amix2, ain

outs aout1, aout2
endin

</CsInstruments>
<CsScore>

; Table 1: A sine wave for the quadrature oscillator.
f 1 0 16384 10 1

; The score.
; p4 = frequency shifter, starting frequency.
; p5 = frequency shifter, ending frequency.
i 1 0 6 -10 10
e

</CsScore>
</CsoundSynthesizer>
```

## Technical History

The use of phase-difference networks in frequency shifters was pioneered by Harald Bode<sup>1</sup>. Bode and Bob Moog provide an excellent description of the implementation and use of a frequency shifter in the analog realm in <sup>2</sup>; this would be an excellent first source for those that wish to explore the possibilities of single sideband modulation. Bernie Hutchins provides more applications of the frequency shifter, as well as a detailed technical analysis<sup>3</sup>. A recent paper by Scott Wardle<sup>4</sup> describes a digital implementation of a frequency shifter, as well as some unique applications.

## References

1. H. Bode, "Solid State Audio Frequency Spectrum Shifter." AES Preprint No. 395 (1965).
2. H. Bode and R.A. Moog, "A High-Accuracy Frequency Shifter for Professional Audio Applications." *Journal of the Audio Engineering Society*, July/August 1972, vol. 20, no. 6, p. 453.
3. B. Hutchins. *Musical Engineer's Handbook* (Ithaca, NY: Electronotes, 1975), ch. 6a.
4. S. Wardle, "A Hilbert-Transformer Frequency Shifter for Audio." Available online at <http://www.i-ua.upf.es/dafx98/papers/>.

## Credits

Author: Sean Costello  
Seattle, Washington  
1999

New in Csound version 3.55

The examples were updated April 2002. Thanks go to Sean Costello for fixing the barberpole example.

# hilbert2

hilbert2 — A Hilbert transformer.

## Description

A DFT-based implementation of a Hilbert transformer.

## Syntax

```
ar1, ar2 hilbert2 asig, ifftsize, ihopsize
```

## Initialisation

*ifftsize* -- DFT analysis size

*ihopsize* -- analysis hopsize

## Performance

*asig* -- input signal

*ar1* -- real output of *asig*

*ar2* -- imag output of *asig*

*hilbert2* is a DFT-based implementation of the Hilbert Transform producing two outputs in quadrature (90 degree phase difference across the spectrum). Unlike the IIR-based *hilbert* opcode, *hilbert2* has a linear frequency response. Given that it employs a streaming algorithm, a delay of *ifftsize* samples will be imposed between input and output.

## Example

Here is an example of the *hilbert2* opcode. It uses the file *hilbert2.csd* [examples/hilbert2.csd].

### Example 400. Example of the *hilbert2* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>

instr 1

asig oscili p4, p5
a1,a2 hilbert2 asig,1024,256
outs a1, a2

endin
```

```
</CsInstruments>  
<CsScore>  
i1 0 10 0.5 440  
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
2017

# hrtfearly

*hrtfearly* — Generates 3D binaural audio with high-fidelity early reflections in a parametric room using a Phase Truncation algorithm.

## Description

This opcode essentially nests the *hrtfmove* opcode in an image model for a user-definable shoebox-shaped room. A default room can be selected, or advanced room parameters can be used. Room surfaces can be controlled with high and low-frequency absorption coefficients and gain factors of a three-band equaliser.

Although valid as a stand alone opcode, *hrtfearly* is designed to work with *hrtfverb* to provide spatially accurate, dynamic binaural reverberation. A number of sources can be processed dynamically using a number of *hrtfearly* instances. All can then be processed with one instance of *hrtfverb*.

## Syntax

```
aleft, aright, irt60low, irt60high, imfp hrtfearly asrc, ksrx, ksry, ksrcz, klstnrx, klstnry, klstnrz
ifilel, ifiler, idefroom [,ifade, isr, iorder, ithreed, kheadrot, iroomx, iroomy, iroomz, iwallh
iwalllow, iwallgain1, iwallgain2, iwallgain3, ifloorhigh, ifloorlow, ifloorgain1, ifloorgain2, \
ifloorgain3, iceilinghigh, iceilinglow, iceilinggain1, iceilinggain2, iceilinggain3]
```

## Initialization

## Initialization

## Initialization

*ifilel* - left HRTF spectral data file.

*ifiler* - right HRTF spectral data file.



### Note

Spectral datafiles (based on the MIT HRTF database) are available in 3 different sampling rates: 44.1, 48 and 96 kHz and are labelled accordingly. Input and processing *sr* should match datafile *sr*. Files should be in the current directory or the SADIR (see *Environment Variables*).



### Note

HRTF Data files for use with *hrtfmove*, *hrtfmove2*, *hrtfstat*, *hrtfearly*, *hrtfverb* were updated for Csound 5.15 and later (the code was updated and is more efficient). Old datafiles are now deprecated.

*idefroom* - default room, medium (1: 10\*10\*3), small (2: 3\*4\*3) or large (3: 20\*25\*7). Wall details (high coef, low coef, gain1, gain2, gain3): .3, .1, .75, .95, .9. Floor: .6, .1, .95, .6, .35. Ceiling: .2, .1, 1, 1, 1. If 0 is entered, optional room parameters will be read.

*ifade* - optional, number of processing buffers for phase change crossfade (default 8). Legal range is 1-24. See *hrtfmove*.

*isr* - optional, default 44.1kHz, legal values: 44100, 48000 and 96000.

*iorder* - optional, order of images processed: higher order: more reflections. Defaults to 1, legal range: 0-4.

*ithreed* - optional, process image sources in three dimensions (1) or two (0: default).

*iroomx* - optional, x room size in metres, will be read if no valid default room is entered (all below parameters behave similarly). Minimum room size is 2\*2\*2.

*iroomy* - optional, y room size.

*iroomz* - optional, z room size.

*iwallhigh* - optional, high frequency wall absorption coefficient (all 4 walls are assumed identical). Absorption coefficients will affect reverb time output.

*iwalllow* - optional, low frequency wall absorption coefficient.

*iwallgain1* - optional, gain on filter centred at 250 Hz (all filters have a Q implying 4 octaves).

*iwallgain2* - optional, as above, centred on 1000 Hz.

*iwallgain3* - optional, as above, centred on 4000 Hz.

*ifloorhigh*, *ifloorlow*, *ifloorgain1*, *ifloorgain2*, *ifloorgain3* - as above for floor.

*iceilinghigh*, *iceilinglow*, *iceilinggain1*, *iceilinggain2*, *iceilinggain3* - as above for ceiling.

*ksrcx* source x location, must be 10 cm inside room. Also, near-field HRTFs are not processed, so source will not change spatially within a 45 cm radius of the listener. These restrictions also apply to location parameters below.

*ksrcy* source y location.

*ksrcz* source z location.

*klstnrx*, *klstnry*, *klstnrz* listener location, as above.

*kheadrot* - optional, angular value for head rotation.

*asrc* - Input/source signal.

*irt60low* - suggested low frequency reverb time for later binaural reverb.

*irt60high* - as above, for high frequency.

*imfp* - mean free path of room, to be used with later reverb.

## Examples

Here is an example of the *hrtfearly* and *hrtfverb* opcodes. It uses the file *hrtfearly.csd* [examples/hrtfearly.csd].

### Example 401. Example of the *hrtfearly* opcode.

```
<CsoundSynthesizer>
<CsOptions>

; Select flags here
; realtime audio out
-o dac
; file output
; -o hrtf.wav
```

```

</CsOptions>
<CsInstruments>

nchnls = 2

gasrc init 0 ;global

instr 1 ;a plucked string, distorted and filtered

    iamp = 15000
    icps = cpspch(p4)

    a1 pluck iamp, icps, icps, 0, 1
    adist distort1 a1, 10, .5, 0, 0
    afilt moogvcf2 adist, 8000, .5
    aout linen afilt, 0, p3, .01

    gasrc = gasrc + aout

endin

instr 10 ;uses output from instr1 as source

    ;simple path for source
    kx line 2, p3, 9

    ;early reflections, room default 1
    aearlyl, aearlyr, irt60low, irt60high, imfp hrtfearly gasrc, kx, 5, 1, 5, 1, 1, "hrtf-44100-left.dat",

    ;later reverb, uses outputs from above
    arevl, arevr, idel hrtfverb gasrc, irt60low, irt60high, "hrtf-44100-left.dat", "hrtf-44100-right.d

    ;delayed and scaled
    alatel delay arevl * .1, idel
    alater delay arevr * .1, idel

    outs aearlyl + alatel, aearlyr + alater

    gasrc = 0

endin

</CsInstruments>
<CsScore>

; Play Instrument 1: a simple arpeggio
i1 0 .2 8.00
i1 + .2 8.04
i1 + .2 8.07
i1 + .2 8.11
i1 + .2 9.02
i1 + 1.5 8.11
i1 + 1.5 8.07
i1 + 1.5 8.04
i1 + 1.5 8.00
i1 + 1.5 7.09
i1 + 4 8.00

; Play Instrument 10 for 13 seconds.
i10 0 13

</CsScore>
</CsSoundSynthesizer>

```

## See Also

*hrtfverb hrtfmove, hrtfmove2, hrtfstat, hrtfer.*

## Credits

Author: Brian Carty  
Maynooth  
2011

# hrtfmove

`hrtfmove` — Generates dynamic 3d binaural audio for headphones using magnitude interpolation and phase truncation.

## Description

This opcode takes a source signal and spatialises it in the 3 dimensional space around a listener by convolving the source with stored head related transfer function (HRTF) based filters.

## Syntax

```
aleft, aright hrtfmove asrc, kAz, kElev, ifilel, ifiler [, imode, ifade, isr]
```

## Initialization

### Initialization

*ifilel* -- left HRTF spectral data file

*ifiler* -- right HRTF spectral data file



#### Note

Spectral datafiles (based on the MIT HRTF database) are available in 3 different sampling rates: 44.1, 48 and 96 khz and are labelled accordingly. Input and processing *sr* should match datafile *sr*. Files should be in the current directory or the SADIR (see *Environment Variables*).



#### Note

HRTF Data files for use with *hrtfmove*, *hrtfmove2*, *hrtfstat*, *hrtfearly*, *hrtfverb* were updated for Csound 5.15 and later (the code was updated and is more efficient). Old datafiles are now deprecated.

*imode* -- optional, default 0 for phase truncation, 1 for minimum phase

*ifade* -- optional, number of processing buffers for phase change crossfade (default 8). Legal range is 1-24. A low value is recommended for complex sources (4 or less: a higher value may make the crossfade audible), a higher value (8 or more: a lower value may make the inconsistency when the filter changes phase values audible) for narrowband sources. Does not effect minimum phase processing.



#### Note

Occasionally fades may overlap (when unnaturally fast/complex trajectories are requested). In this case, a warning will be printed. Use a smaller crossfade or slightly change trajectory to avoid any possible inconsistencies that may arise.

*isr* - optional, default 44.1kHz, legal values: 44100, 48000 and 96000.

*kAz* -- azimuth value in degrees. Positive values represent position on the right, negative values are positions on the left.

*kElev* -- elevation value in degrees. Positive values represent position above horizontal, negative values are positions below horizontal (min -40).

Artifact-free user-defined trajectories are made possible using an interpolation algorithm based on spectral magnitude interpolation and phase truncation. Crossfades are implemented to minimise/eliminate any inconsistencies caused by updating phase values. These crossfades are performed over a user definable number of convolution processing buffers. Complex sources may only need to crossfade over 1 buffer; narrow band sources may need several. The opcode also offers minimum phase based processing, a more traditional and complex method. In this mode, the hrtf filters used are reduced to minimum phase representations and the interpolation process then uses the relationship between minimum phase magnitude and phase spectra. Interaural time difference, which is inherent to the phase truncation process, is reintroduced in the minimum phase process using variable delay lines.

## Examples

Here is an example of the *hrtfmove* opcode. It uses the file *hrtfmove.csd* [examples/hrtfmove.csd].

### Example 402. Example of the *hrtfmove* opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select flags here
; realtime audio out
-o dac
; For Non-realtime output leave only the line below:
;-o hrtf.wav
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

gasrc init 0

instr 1 ;a plucked string

    kamp = p4
    kcps = cpspch(p5)
    icps = cpspch(p5)

    a1 pluck kamp, kcps, icps, 0, 1

    gasrc = a1

endin

instr 10 ;uses output from instr1 as source

    kaz linseg 0, p3, 720 ;2 full rotations

    aleft,aright hrtfmove gasrc, kaz,0, "hrtf-44100-left.dat", "hrtf-44100-right.dat"

    outs aleft, aright

endin

</CsInstruments>
<CsScore>
```

```
; Play Instrument 1: a simple arpeggio
i1 0 .2 15000 8.00
i1 + .2 15000 8.04
i1 + .2 15000 8.07
i1 + .2 15000 8.11
i1 + .2 15000 9.02
i1 + 1.5 15000 8.11
i1 + 1.5 15000 8.07
i1 + 1.5 15000 8.04
i1 + 1.5 15000 8.00
i1 + 1.5 15000 7.09
i1 + 1.5 15000 8.00

; Play Instrument 10 for 10 seconds.
i10 0 10

</CsScore>
</CsoundSynthesizer>
```

## See Also

*hrtfmove2, hrtfstat, hrtfer.*

More information on this opcode: <http://www.csoundjournal.com/issue9/newHRTFOpcodes.html> , written by Brian Carty

## Credits

Author: Brian Carty  
Maynooth  
2008

# hrtfmove2

`hrtfmove2` — Generates dynamic 3d binaural audio for headphones using a Woodworth based spherical head model with improved low frequency phase accuracy.

## Description

This opcode takes a source signal and spatialises it in the 3 dimensional space around a listener using head related transfer function (HRTF) based filters.

## Syntax

```
aleft, aright hrtfmove2 asrc, kAz, kElev, ifilel, ifiler [,ioverlap, iradius, isr]
```

## Initialization

*ifilel* -- left HRTF spectral data file

*ifiler* -- right HRTF spectral data file



### Note

Spectral datafiles (based on the MIT HRTF database) are available in 3 different sampling rates: 44.1, 48 and 96 khz and are labelled accordingly. Input and processing *sr* should match datafile *sr*. Files should be in the current directory or the SADIR (see *Environment Variables*).



### Note

HRTF Data files for use with *hrtfmove*, *hrtfmove2*, *hrtfstat*, *hrtfearly*, *hrtfverb* were updated for Csound 5.15 and later (the code was updated and is more efficient). Old datafiles are now deprecated.

*ioverlap* -- optional, number of overlaps for STFT processing (default 4). See STFT section of manual.

*iradius* -- optional, head radius used for phase spectra calculation in centimeters (default 9.0)

*isr* - optional, default 44.1kHz, legal values: 44100, 48000 and 96000.

## Performance

*asrc* -- Input/source signal.

*kAz* -- azimuth value in degrees. Positive values represent position on the right, negative values are positions on the left.

*kElev* -- elevation value in degrees. Positive values represent position above horizontal, negative values are positions below horizontal (min -40).

Artifact-free user-defined trajectories are made possible using an interpolation algorithm based on spectral magnitude interpolation and a derived phase spectrum based on the Woodworth spherical head model.

Accuracy is increased for the data set provided by extracting and applying a frequency dependent scaling factor to the phase spectra, leading to a more precise low frequency interaural time difference. Users can control head radius for the phase derivation, allowing a crude level of individualisation. The dynamic source version of the opcode uses a Short Time Fourier Transform algorithm to avoid artefacts caused by derived phase spectra changes. STFT processing means this opcode is more computationally intensive than *hrtfmove* using phase truncation, but phase is constantly updated by *hrtfmove2*.

## Examples

Here is an example of the *hrtfmove2* opcode. It uses the file *hrtfmove2.csd* [examples/hrtfmove2.csd].

### Example 403. Example of the *hrtfmove2* opcode.

```

<CsoundSynthesizer>
<CsOptions>
; Select flags here
; realtime audio out
-o dac
; For Non-realtime ouput leave only the line below:
; -o hrtf.wav
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

gasrc init 0

instr 1 ;a plucked string

  kamp = p4
  kcps = cpspch(p5)
  icps = cpspch(p5)

  a1 pluck kamp, kcps, icps, 0, 1

  gasrc = a1

endin

instr 10 ;uses output from instr1 as source

  kaz linseg 0, p3, 720 ;2 full rotations

  aleft,aright hrtfmove2 gasrc, kaz,0, "hrtf-44100-left.dat", "hrtf-44100-right.dat"

  outs aleft, aright

endin

</CsInstruments>
<CsScore>

; Play Instrument 1: a simple arpeggio
i1 0 .2 15000 8.00
i1 + .2 15000 8.04
i1 + .2 15000 8.07
i1 + .2 15000 8.11
i1 + .2 15000 9.02
i1 + 1.5 15000 8.11

```

```
i1 + 1.5 15000 8.07
i1 + 1.5 15000 8.04
i1 + 1.5 15000 8.00
i1 + 1.5 15000 7.09
i1 + 1.5 15000 8.00

; Play Instrument 10 for 10 seconds.
i10 0 10

</CsScore>
</CsoundSynthesizer>
```

## See Also

*hrtfmove, hrtfstat, hrtfer.*

More information on this opcode: <http://www.csoundjournal.com/issue9/newHRTFOpcodes.html> , written by Brian Carty

## Credits

Author: Brian Carty  
Maynooth  
2008

# hrtfverb

*hrtfverb* — A binaural, dynamic FDN based diffuse-field reverberator. The opcode works independently as an efficient, flexible reverberator.

## Description

A frequency-dependent, efficient reverberant field is created based on low and high frequency desired reverb times. The opcode is designed to work with *hrtfearly*, ideally using its outputs as inputs. However, *hrtfverb* can be used as a standalone tool. Stability is enforced.

It is, however, designed for use with *hrtfearly* to provide spatially accurate reverberation with user definable source trajectories. Accurate interaural coherence is also provided.

## Syntax

```
aleft, aright, idel hrtfverb asrc, ilowrt60, ihighrt60, ifilel, ifiler [,isr, imfp, iorder]
```

## Initialization

## Initialization

## Initialization

*ilowrt60* - low frequency reverb time.

*ihighrt60* - high frequency reverb time.

*ifilel* - left HRTF spectral data file.

*ifiler* - right HRTF spectral data file.



### Note

Spectral datafiles (based on the MIT HRTF database) are available in 3 different sampling rates: 44.1, 48 and 96 kHz and are labelled accordingly. Input and processing *sr* should match datafile *sr*. Files should be in the current directory or the SADIR (see *Environment Variables*).



### Note

HRTF Data files for use with *hrtfmove*, *hrtfmove2*, *hrtfstat*, *hrtfverb*, *hrtfverb* were updated for Csound 5.15 and later (the code was updated and is more efficient). Old datafiles are now deprecated.

*isr* - optional, default 44.1kHz, legal values: 44100, 48000 and 96000.

*imfp* - optional, mean free path, defaults to that of a medium room. If used with *hrtfearly*, the mean free path of the room can be used to calculate the appropriate delay for the later reverb. Legal range: the mean free path of the smallest room allowed by *hrtfearly* (0.003876) 1.

*iorder* - optional, order of early reflection processing. If used with *hrtfearly*, the order of early reflections can be used to calculate the appropriate delay on the later reverb.

*asrc* - Input/source signal.

*idel* - if used with *hrtfearly*, the appropriate delay for the later reverb, based on the room and order of processing.

## Examples

Here is an example of the *hrtfearly* and *hrtfreverb* opcodes. It uses the file *hrtfearly.csd* [examples/hrtfearly.csd].

### Example 404. Example of the *hrtfearly* opcode.

```
<CsoundSynthesizer>
<CsOptions>

; Select flags here
; realtime audio out
-o dac
; file output
; -o hrtf.wav

</CsOptions>
<CsInstruments>

nchnls = 2

gasrc init 0 ;global

instr 1 ;a plucked string, distorted and filtered

    iamp = 15000
    icps = cpspch(p4)

    a1 pluck iamp, icps, icps, 0, 1
    adist distort1 a1, 10, .5, 0, 0
    afilt moogvcf2 adist, 8000, .5
    aout linen afilt, 0, p3, .01

    gasrc = gasrc + aout

endin

instr 10 ;uses output from instr1 as source

;simple path for source
kx line 2, p3, 9

;early reflections, room default 1
aearlyl,aearlyr, irt60low, irt60high, imfp hrtfearly gasrc, kx, 5, 1, 5, 1, 1, "hrtf-44100-left.dat",

;later reverb, uses outputs from above
arevl, arevr, idel hrtfreverb gasrc, irt60low, irt60high, "hrtf-44100-left.dat", "hrtf-44100-right.da

;delayed and scaled
alatel delay arevl * .1, idel
alater delay arevr * .1, idel

outs aearlyl + alatel, aearlyr + alater
```

```
gasrc = 0

endin

</CsInstruments>
<CsScore>

; Play Instrument 1: a simple arpeggio
i1 0 .2 8.00
i1 + .2 8.04
i1 + .2 8.07
i1 + .2 8.11
i1 + .2 9.02
i1 + 1.5 8.11
i1 + 1.5 8.07
i1 + 1.5 8.04
i1 + 1.5 8.00
i1 + 1.5 7.09
i1 + 4 8.00

; Play Instrument 10 for 13 seconds.
i10 0 13

</CsScore>
</CsoundSynthesizer>
```

## See Also

*hrtfearly hrtfmove, hrtfmove2, hrtfstat, hrtfer.*

## Credits

Author: Brian Carty  
Maynooth  
2011

## hrtfstat

`hrtfstat` — Generates static 3d binaural audio for headphones using a Woodworth based spherical head model with improved low frequency phase accuracy.

## Description

This opcode takes a source signal and spatialises it in the 3 dimensional space around a listener using head related transfer function (HRTF) based filters. It produces a static output (azimuth and elevation parameters are `i-rate`), because a static source allows much more efficient processing than `hrtfmove` and `hrtfmove2`.

## Syntax

```
aleft, aright hrtfstat asrc, iAz, iElev, ifilel, ifiler [,iradius, isr]
```

## Initialization

`iAz` -- azimuth value in degrees. Positive values represent position on the right, negative values are positions on the left.

`iElev` -- elevation value in degrees. Positive values represent position above horizontal, negative values are positions below horizontal (min -40).

`ifilel` -- left HRTF spectral data file

`ifiler` -- right HRTF spectral data file



### Note

Spectral datafiles (based on the MIT HRTF database) are available in 3 different sampling rates: 44.1, 48 and 96 khz and are labelled accordingly. Input and processing `sr` should match datafile `sr`. Files should be in the current directory or the `SADIR` (see *Environment Variables*).



### Note

HRTF Data files for use with `hrtfmove`, `hrtfmove2`, `hrtfstat`, `hrtfearly`, `hrtfverb` were updated for Csound 5.15 and later (the code was updated and is more efficient). Old datafiles are now deprecated.

`iradius` -- optional, head radius used for phase spectra calculation in centimeters (default 9.0)

`isr` - optional (default 44.1kHz). Legal values are 44100, 48000 and 96000.

## Performance

Artifact-free user-defined static spatialisation is made possible using an interpolation algorithm based on spectral magnitude interpolation and a derived phase based on the Woodworth spherical head model. Accuracy is increased for the data set provided by extracting and applying a frequency dependent scaling factor to the phase spectra, leading to a more precise low frequency interaural time difference. Users can control head radius for the phase derivation, allowing a crude level of individualisation. The static source

version of the opcode uses overlap add convolution (it does not need STFT processing, see *hrtfmove2*), and is thus considerably more efficient than *hrtfmove2* or *hrtfmove*, but cannot generate moving sources.

## Examples

Here is an example of the *hrtfstat* opcode. It uses the file *hrtfstat.csd* [examples/hrtfstat.csd].

### Example 405. Example of the *hrtfstat* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o hrtfstat.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gasrc init 0

instr 1 ;a plucked string

kamp = p4
kcps = cpspch(p5)
icps = cpspch(p5)
a1 pluck kamp, kcps, icps, 0, 1

gasrc = a1

endin

instr 10;uses output from instr1 as source

aleft,right hrtfstat gasrc, 90,0, "hrtf-44100-left.dat", "hrtf-44100-right.dat"
outs      aleft, right

clear gasrc
endin

</CsInstruments>
<CsScore>

i1 0 2 .7 8.00 ; Play Instrument 1: a plucked string
i1 .5 2 .7 8.00
i1 1 2 .7 8.00
i1 2 2 .7 7.00

i10 0 12 ; Play Instrument 10 for 2 seconds.

</CsScore>
</CsoundSynthesizer>
```

Here is another example of the *hrtfstat* opcode. It uses the file *hrtfstat-2.csd* [examples/hrtfstat-2.csd], and *Church.wav* [examples/Church.wav], which is a looped sample.

## Example 406. Example two of the hrtfstat opcode

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o hrtfstat-2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

iAz = p4
iElev = p5

itim = ftlptim(1)
; transeg a dur ty b dur ty c dur ty d
kamp transeg 0, p3*.1, 0, .9, p3*.3, -3, .5, p3*.3, -2, 0
ain loscil3 kamp, 50, 1
aleft,aright hrtfstat ain, iAz, iElev, "hrtf-44100-left.dat","hrtf-44100-right.dat"
outs aleft, aright

endin
</CsInstruments>
<CsScore>
f 1 0 0 1 "Church.wav" 0 0 0 ;Csound computes tablesize

; Azim Elev
i1 0 7 90 0 ;to the right
i1 3 7 -90 -40 ;to the left and below
i1 6 7 180 90 ;behind and up
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*hrtfmove, hrtfmove2, hrtfer.*

More information on this opcode: <http://www.csoundjournal.com/issue9/newHRTFOpcodes.html> , written by Brian Carty

## Credits

Author: Brian Carty  
Maynooth  
2008

# hsboscil

hsboscil — An oscillator which takes tonality and brightness as arguments.

## Description

An oscillator which takes tonality and brightness as arguments, relative to a base frequency.

## Syntax

```
ares hsboscil kamp, ktone, kbrite, ibasfreq, iwfn, ioctfn \  
    [, ioctcnt] [, iphs]
```

## Initialization

*ibasfreq* -- base frequency to which tonality and brightness are relative

*iwfn* -- function table of the waveform, usually a sine

*ioctfn* -- function table used for weighting the octaves, usually something like:

```
f1 0 1024 -19 1 0.5 270 0.5
```

*ioctcnt* (optional) -- number of octaves used for brightness blending. Must be in the range 2 to 10. Default is 3.

*iphs* (optional, default=0) -- initial phase of the oscillator. If *iphs* = -1, initialization is skipped.

## Performance

*kamp* -- amplitude of note

*ktone* -- cyclic tonality parameter relative to *ibasfreq* in logarithmic octave, range 0 to 1, values > 1 can be used, and are internally reduced to  $frac(ktone)$ .

*kbrite* -- brightness parameter relative to *ibasfreq*, achieved by weighting *ioctcnt* octaves. It is scaled in such a way, that a value of 0 corresponds to the original value of *ibasfreq*, 1 corresponds to one octave above *ibasfreq*, -2 corresponds to two octaves below *ibasfreq*, etc. *kbrite* may be fractional.

*hsboscil* takes tonality and brightness as arguments, relative to a base frequency (*ibasfreq*). Tonality is a cyclic parameter in the logarithmic octave, brightness is realized by mixing multiple weighted octaves. It is useful when tone space is understood in a concept of polar coordinates.

Making *ktone* a line, and *kbrite* a constant, produces Risset's glissando.

Oscillator table *iwfn* is always read interpolated. Performance time requires about *ioctcnt* \* *oscili*.

## Examples

Here is an example of the hsboscil opcode. It uses the file *hsboscil.csd* [examples/hsboscil.csd].

**Example 407. Example of the hsboscil opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o hsboscil.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

; synth waveform
giwave ftgen 1, 0, 1024, 10, 1, 1, 1, 1
; blending window
giblend ftgen 2, 0, 1024, -19, 1, 0.5, 270, 0.5

instr 1 ; produces Risset's glissando.

    kamp = .4
    kbrite = 0.3
    ibasfreq = 200
    ioctcnt = 5

    ; Change ktone linearly from 0 to 1,
    ; over the period defined by p3.
    ktone line 0, p3, 1

asig hsboscil kamp, ktone, kbrite, ibasfreq, giwave, giblend, ioctcnt
    outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 10
e
</CsScore>
</CsoundSynthesizer>

```

Here is an example of the hsboscil opcode in a MIDI instrument. It uses the file *hsboscil\_midi.csd* [examples/hsboscil\_midi.csd].

**Example 408. Example of the hsboscil opcode in a MIDI instrument.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out
-odac      -M0 ;;realtime audio out and realtime MIDI in
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; -o hsboscil_midi.wav -W ;; for file output any platform

```

```
</CsOptions>
<CsInstruments>

  sr = 44100
  ksmps = 32
  nchnls = 2
  0dbfs = 1

  ; synth waveform
  giwave ftgen 1, 0, 1024, 10, 1, 1, 1, 1
  ; blending window
  giblend ftgen 2, 0, 1024, -19, 1, 0.5, 270, 0.5

  instr 1

  ibase = cpsoct(6)
  ioctcnt = 5

  ; all octaves sound alike.
  itona octmidi
  ; velocity is mapped to brightness
  ibrite ampmidi 4

  ; Map an exponential envelope for the amplitude.
  kenv expon .8, 1, .01
  asig hsboscil kenv, itona, ibrite, ibase, giwave, giblend, ioctcnt
  outs asig, asig

  endin
</CsInstruments>
<CsScore>

  i 1 0 30 ; play for 30 seconds
  e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Peter Neubäcker  
Munich, Germany  
August, 1999

New in Csound version 3.58

# hvs1

`hvs1` — Allows one-dimensional Hyper Vectorial Synthesis (HVS) controlled by externally-updated k-variables.

## Description

*hvs1* allows one-dimensional Hyper Vectorial Synthesis (HVS) controlled by externally-updated k-variables.

## Syntax

```
hvs1 kx, inumParms, inumPointsX, iOutTab, iPositionsTab, iSnapTab [, iConfigTab]
```

## Initialization

*inumParms* - number of parameters controlled by the HVS. Each HVS snapshot is made up of *inumParms* elements.

*inumPointsX* - number of points that each dimension of the HVS cube (or square in case of two-dimensional HVS; or line in case of one-dimensional HVS) is made up.

*iOutTab* - number of the table receiving the set of output-parameter instant values of the HVS. The total amount of parameters is defined by the *inumParms* argument.

*iPositionsTab* – a table filled with the individual positions of snapshots in the HVS matrix (see below for more information).

*iSnapTab* – a table filled with all the snapshots. Each snapshot is made up of a set of parameter values. The amount of elements contained in each snapshots is specified by the *inumParms* argument. The set of elements of each snapshot follows (and is adjacent) to the previous one in this table. So the total size of this table should be  $\geq$  to *inumParms* multiplied the number of snapshots you intend to store for the HVS.

*iConfigTab* – (optional) a table containing the behavior of the HVS for each parameter. If the value of *iConfigTab* is zero (default), this argument is ignored, meaning that each parameter is treated with linear interpolation by the HVS. If *iConfigTab* is different than zero, then it must refer to an existing table whose contents are in its turn referring to a particular kind of interpolation. In this table, a value of -1 indicates that corresponding parameter is leaved unchanged (ignored) by the HVS; a value of zero indicates that corresponding parameter is treated with linear-interpolation; each other values must be integer numbers indicating an existing table filled with a shape which will determine the kind of special interpolation to be used (table-based interpolation).

## Performance

*kx* - these are externally-modified variables which controls the motion of the pointer in the HVS matrix cube (or square or line in case of HVS matrices made up of less than 3 dimensions). The range of these input arguments must be 0 to 1.

Hyper Vectorial Synthesis is a technique that allows control of a huge set of parameters by using a simple and global approach. The key concepts of the HVS are:

The set of HVS parameters, whose amount is fixed and defined by the *inumParms* argument. During the HVS performance, all these parameters are variant and can be applied to any sound synthesis technique, as

well as to any global control for algorithmic composition and any other kind of level. The user must previously define several sets of fixed values for each HVS parameter, each set corresponding to a determinate synthesis configuration. Each set of values is called snapshot, and can be considered as the coordinates of a bound of a multi-dimensional space. The HVS consists on moving a point in this multi-dimensional space (by using a special motion pointer, see below), according and inside the bounds defined by the snapshots. You can fix any amount of HVS parameters (each parameter being a dimension of the multi-dimensional space), even a huge number, the limit only depends on the processing power (and the memory) of your computer and on the complexity of the sound-synthesis you will use.

The HVS cube (or square or line). This is the matrix (of 3, 2 or 1 dimensions, according to the `hvs` opcode you intend to use) of “mainstays” (or pivot) points of HVS. The total amount of pivot-points depends on the value of the `inumPointsX`, `inumPointsY` and `inumPointsZ` arguments. In the case of a 3-dimensional HVS matrix you can define, for instance, 3 points for the X dimension, 5 for the Y dimension and 2 for the Z dimension. In this case, the total number of pivot-points is  $3 * 5 * 2 = 30$ . With this set of pivot points, the cube is divided into smaller cubed zones each one bounded by eight nearby points. Each point is numbered. The numeral order of these points is established in the following way: number zero is the first point, number 1 the second and so on. Assuming you are using a 3-dimensional HVS cube having the number of points above mentioned (i.e. 3, 5 and 2 respectively for the X, Y and Z axis), the first point (point zero) is the upper-left-front vertex of the cube, by facing the XY plane of the cube. The second point is the middle point of the upper front edge of the cube and so on. You can refer to the figure below in order to understand how the numeral order of the pivot-points proceeds:

For the 2-dimensional HVS, it is the same, by only omitting the rear cube face, so each zone is bounded by 4 pivot-points instead of 8. For the 1-dimensional HVS, the whole thing is even simpler because it is a line with the pivot-points proceeding from left to right. Each point is coupled with a snapshot.

Snapshot order, as stored into the `iSnapTab`, can or cannot follow the order of the pivot-points numbers. In fact it is possible to alter this order by means the `iPositionsTab`, a table that remaps the position of each snapshot in relation to the pivot points. The `iPositionsTab` is made up of the positions of the snapshots (contained in the `iSnapTab`) in the two-dimensional grid. Each subsequent element is actually a pointer representing the position in the `iSnapTab`. For example, in a 2-dimensional HVS matrix such as the following (in this case having `inumPointsX = 3` and `inumPointsY = 5`):

**Table 9.**

5	7	1
3	4	9
6	2	0
4	1	3
8	2	7

These numbers (to be stored in the `iSnapTab` table by using, for instance, the GEN02 function generator) represents the snapshot position within the grid (in this case a 3x5 matrix). So, the first element 5, has index zero and represents the 6th (element zero is the first) snapshot contained in the `iSnapTab`, the second element 7 represents the 8th element of `iSnapTab` and so on. Summing up, the vertices of each zone (a cubed zone is delimited by 8 vertices; a squared zone by 4 vertices and a linear zone by 2 points) are coupled with a determinate snapshot, whose number is remapped by the `iSnapTab`.

Output values of the HVS are influenced by the motion pointer, a point whose position, in the HVS cube (or square or segment) is determined by the `kx`, `ky` and `kz` arguments. The values of these arguments, which must be in the 0 to 1 range, are externally set by the user. The output values, whose amount is equal to the `inumParms` argument, are stored in the `iOutTab`, a table that must be already allocated by the user, and must be at least `inumParms` size. In what way the motion pointer influences the output? Well, when the motion

pointer falls in a determinate cubed zone, delimited, for instance, by 8 vertices (or pivot points), we assume that each vertex has associated a different snapshot (i.e. a set of *inumParms* values), well, the output will be the weighted average value of the 8 vertices, calculated according on the distance of the motion pointer from each of the 8 vertices. In the case of a default behavior, when the *iConfigTab* argument is not set, the exact output is calculated by using linear interpolation which is applied to each different parameter of the HVS. Anyway, it is possible to influence this behavior by setting the *iConfigTab* argument to a number of a table whose contents can affect one or more HVS parameters. The *iConfigTab* table elements are associated to each HVS parameter and their values affect the HVS output in the following way:

- If *iConfigTab* is equal to -1, corresponding output is skipped, i.e. the element is not calculated, leaving corresponding element value in the *iOutTab* unchanged;
- If *iConfigTab* is equal to zero, then the normal HVS output is calculated (by using weighted average of the nearest vertex of current zone where it falls the motion pointer);
- If *iConfigTab* element is equal to an integer number > zero, then the contents of a table having that number is used as a shape of a table-based interpolation.

## Examples

Here is an example of the *hvs1* opcode. It uses the file *hvs1.csd* [examples/hvs1.csd].

### Example 409. Example of the *hvs1* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac        -iadc     -d      ;;RT audio I/O
</CsOptions>
<CsInstruments>

sr=44100
ksmps=100
nchnls=2
0dbfs = 1

; Example by Gabriel Maldonado and Andres Cabrera

ginumLinesX  init 16
ginumParms   init 3

giOutTab  ftgen 5,0,8, -2,      0
giPosTab  ftgen 6,0,32, -2,    3,2,1,0,4,5,6,7,8,9,10, 11, 15, 14, 13, 12
giSnapTab ftgen 8,0,64, -2,    1,1,1,  2,0,0,  3,2,0,  2,2,2,  5,2,1,  2,3,4,  6,1,7,  0,0,0, \
1,3,5,   3,4,4,  1,5,8,  1,1,5,  4,3,2,  3,4,5,  7,6,5,  7,8,9

tb0_init giOutTab

      FLpanel "hsv1",500,100,10,10,0
gk1,ih1 FLslider "X", 0,1,  0,5, -1, 400,30, 50,20
      FLpanel_end
      FLrun

instr 1
;      kx,   inumParms, inumPointsX, iOutTab, iPosTab, iSnapTab [, iConfigTab]
hvs1    gk1,  ginumParms, ginumLinesX, giOutTab, giPosTab, giSnapTab ;, iConfigTab
```

```
k0 init 0
k1 init 1
k2 init 2

printk2 tb0(k0)
printk2 tb0(k1), 10
printk2 tb0(k2), 20

aosc1 oscil tb0(k0)/20, tb0(k1)*100 + 200, 1
aosc2 oscil tb0(k1)/20, tb0(k2)*100 + 200, 1
aosc3 oscil tb0(k2)/20, tb0(k0)*100 + 200, 1
aosc4 oscil tb0(k1)/20, tb0(k0)*100 + 200, 1
aosc5 oscil tb0(k2)/20, tb0(k1)*100 + 200, 1
aosc6 oscil tb0(k0)/20, tb0(k2)*100 + 200, 1

outs aosc1 + aosc2 + aosc3, aosc4 + aosc5 + aosc6
endin

</CsInstruments>
<CsScore>

f1 0 1024 10 1
f0 3600
i1 0 3600

</CsScore>
</CsoundSynthesizer>
```

## See Also

*hvs2, hvs3, vphaseseg*

## Credits

Author: Gabriel Maldonado

New in version 5.06

## hvs2

**hvs2** — Allows two-dimensional Hyper Vectorial Synthesis (HVS) controlled by externally-updated k-variables.

### Description

*hvs2* allows two-dimensional Hyper Vectorial Synthesis (HVS) controlled by externally-updated k-variables.

### Syntax

**hvs2** *kx*, *ky*, *inumParms*, *inumPointsX*, *inumPointsY*, *iOutTab*, *iPositionsTab*, *iSnapTab* [, *iConfigTab*]

### Initialization

*inumParms* - number of parameters controlled by the HVS. Each HVS snapshot is made up of *inumParms* elements.

*inumPointsX*, *inumPointsY* - number of points that each dimension of the HVS cube (or square in case of two-dimensional HVS; or line in case of one-dimensional HVS) is made up.

*iOutTab* - number of the table receiving the set of output-parameter instant values of the HVS. The total amount of parameters is defined by the *inumParms* argument.

*iPositionsTab* – a table filled with the individual positions of snapshots in the HVS matrix (see below for more information).

*iSnapTab* – a table filled with all the snapshots. Each snapshot is made up of a set of parameter values. The amount of elements contained in each snapshots is specified by the *inumParms* argument. The set of elements of each snapshot follows (and is adjacent) to the previous one in this table. So the total size of this table should be  $\geq$  to *inumParms* multiplied the number of snapshots you intend to store for the HVS.

*iConfigTab* – (optional) a table containing the behavior of the HVS for each parameter. If the value of *iConfigTab* is zero (default), this argument is ignored, meaning that each parameter is treated with linear interpolation by the HVS. If *iConfigTab* is different than zero, then it must refer to an existing table whose contents are in its turn referring to a particular kind of interpolation. In this table, a value of -1 indicates that corresponding parameter is leaved unchanged (ignored) by the HVS; a value of zero indicates that corresponding parameter is treated with linear-interpolation; each other values must be integer numbers indicating an existing table filled with a shape which will determine the kind of special interpolation to be used (table-based interpolation).

### Performance

*kx*, *ky* - these are externally-modified variables which controls the motion of the pointer in the HVS matrix cube (or square or line in case of HVS matrices made up of less than 3 dimensions). The range of these input arguments must be 0 to 1.

Hyper Vectorial Synthesis is a technique that allows control of a huge set of parameters by using a simple and global approach. The key concepts of the HVS are:

The set of HVS parameters, whose amount is fixed and defined by the *inumParms* argument. During the HVS performance, all these parameters are variant and can be applied to any sound synthesis technique, as well as to any global control for algorithmic composition and any other kind of level. The user must previ-

ously define several sets of fixed values for each HVS parameter, each set corresponding to a determinate synthesis configuration. Each set of values is called snapshot, and can be considered as the coordinates of a bound of a multi-dimensional space. The HVS consists on moving a point in this multi-dimensional space (by using a special motion pointer, see below), according and inside the bounds defined by the snapshots. You can fix any amount of HVS parameters (each parameter being a dimension of the multi-dimensional space), even a huge number, the limit only depends on the processing power (and the memory) of your computer and on the complexity of the sound-synthesis you will use.

The HVS cube (or square or line). This is the matrix (of 3, 2 or 1 dimensions, according to the `hvs` opcode you intend to use) of “mainstays” (or pivot) points of HVS. The total amount of pivot-points depends on the value of the `inumPointsX`, `inumPointsY` and `inumPointsZ` arguments. In the case of a 3-dimensional HVS matrix you can define, for instance, 3 points for the X dimension, 5 for the Y dimension and 2 for the Z dimension. In this case, the total number of pivot-points is  $3 * 5 * 2 = 30$ . With this set of pivot points, the cube is divided into smaller cubed zones each one bounded by eight nearby points. Each point is numbered. The numeral order of these points is established in the following way: number zero is the first point, number 1 the second and so on. Assuming you are using a 3-dimensional HVS cube having the number of points above mentioned (i.e. 3, 5 and 2 respectively for the X, Y and Z axis), the first point (point zero) is the upper-left-front vertex of the cube, by facing the XY plane of the cube. The second point is the middle point of the upper front edge of the cube and so on. You can refer to the figure below in order to understand how the numeral order of the pivot-points proceeds:

For the 2-dimensional HVS, it is the same, by only omitting the rear cube face, so each zone is bounded by 4 pivot-points instead of 8. For the 1-dimensional HVS, the whole thing is even simpler because it is a line with the pivot-points proceeding from left to right. Each point is coupled with a snapshot.

Snapshot order, as stored into the `iSnapTab`, can or cannot follow the order of the pivot-points numbers. In fact it is possible to alter this order by means the `iPositionsTab`, a table that remaps the position of each snapshot in relation to the pivot points. The `iPositionsTab` is made up of the positions of the snapshots (contained in the `iSnapTab`) in the two-dimensional grid. Each subsequent element is actually a pointer representing the position in the `iSnapTab`. For example, in a 2-dimensional HVS matrix such as the following (in this case having `inumPointsX = 3` and `inumPointsY = 5`:

**Table 10.**

5	7	1
3	4	9
6	2	0
4	1	3
8	2	7

These numbers (to be stored in the `iSnapTab` table by using, for instance, the GEN02 function generator) represents the snapshot position within the grid (in this case a 3x5 matrix). So, the first element 5, has index zero and represents the 6th (element zero is the first) snapshot contained in the `iSnapTab`, the second element 7 represents the 8th element of `iSnapTab` and so on. Summing up, the vertices of each zone (a cubed zone is delimited by 8 vertices; a squared zone by 4 vertices and a linear zone by 2 points) are coupled with a determinate snapshot, whose number is remapped by the `iSnapTab`.

Output values of the HVS are influenced by the motion pointer, a point whose position, in the HVS cube (or square or segment) is determined by the `kx`, `ky` and `kz` arguments. The values of these arguments, which must be in the 0 to 1 range, are externally set by the user. The output values, whose amount is equal to the `inumParms` argument, are stored in the `iOutTab`, a table that must be already allocated by the user, and must be at least `inumParms` size. In what way the motion pointer influences the output? Well, when the motion pointer falls in a determinate cubed zone, delimited, for instance, by 8 vertices (or pivot points), we assume that each vertex has associated a different snapshot (i.e. a set of `inumParms` values), well, the output will

be the weighted average value of the 8 vertices, calculated according on the distance of the motion pointer from each of the 8 vertices. In the case of a default behavior, when the *iConfigTab* argument is not set, the exact output is calculated by using linear interpolation which is applied to each different parameter of the HVS. Anyway, it is possible to influence this behavior by setting the *iConfigTab* argument to a number of a table whose contents can affect one or more HVS parameters. The *iConfigTab* table elements are associated to each HVS parameter and their values affect the HVS output in the following way:

- If *iConfigTab* is equal to -1, corresponding output is skipped, i.e. the element is not calculated, leaving corresponding element value in the *iOutTab* unchanged;
- If *iConfigTab* is equal to zero, then the normal HVS output is calculated (by using weighted average of the nearest vertex of current zone where it falls the motion pointer);
- If *iConfigTab* element is equal to an integer number > zero, then the contents of a table having that number is used as a shape of a table-based interpolation.

## Examples

Here is an example of the hvs2 opcode. It uses the file *hvs2.csd* [examples/hvs2.csd].

### Example 410. Example of the hvs2 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      ;;RT audio I/O
</CsOptions>
<CsInstruments>

sr=44100
ksmps=100
nchnls=2

0dbfs = 1

ginumLinesX init 4
ginumLinesY init 4
ginumParms init 3

giOutTab ftgen 5,0,8, -2,      0
giPosTab ftgen 6,0,32, -2,    3,2,1,0,4,5,6,7,8,9,10, 11, 15, 14, 13, 12
giSnapTab ftgen 8,0,64, -2,    1,1,1,  2,0,0,  3,2,0,  2,2,2,  5,2,1,  2,3,4,  6,1,7,  0,0,0, \
1,3,5,  3,4,4,  1,5,8,  1,1,5,  4,3,2,  3,4,5,  7,6,5,  7,8,9

tb0_init giOutTab

FLpanel "Prova HVS2",600,400,10,100,0

gk1,  gk2,  ih1, ih2  FLjoy  "HVS controller XY", 0,  1,  1,  0,  0,  0,  -1,

; *ihandle,                      *numlinesX,  *numlinesY, *iwidth, *iheight, *ix, *iy,*image;
gihandle FLhvsBox ginumLinesX,  ginumLinesY, 300, 300, 300, 50, 1

FLpanel_end
FLrun

instr 1
```

```

; Smooth control signals to avoid clicks
kx portk gk1, 0.02
ky portk gk2, 0.02

;
      kx, ky, inumParms, inumlinesX, inumlinesY, iOutTab, iPosTab, iSnapTab [, iConfigT
hvs2 kx, ky, ginumParms, ginumLinesX, ginumLinesY, giOutTab, giPosTab, giSnapTab ;, iConfigT

;
      *kx, *ky, *ihandle;
FLhvsBoxSetValue gk1, gk2, gihandle

k0 init 0
k1 init 1
k2 init 2

printk2 tb0(k0)
printk2 tb0(k1), 10
printk2 tb0(k2), 20

      kris init 0.003
      kdur init 0.02
      kdec init 0.007

; Make parameters of synthesis depend on the table values produced by hvs
ares1 fof 0.2, tb0(k0)*100 + 50, tb0(k1)*100 + 200, 0, tb0(k2) * 10 + 50, 0.003, 0.02, 0.007, 20, \
      1, 2, p3
ares2 fof 0.2, tb0(k1)*100 + 50, tb0(k2)*100 + 200, 0, tb0(k0) * 10 + 50, 0.003, 0.02, 0.007, 20, \
      1, 2, p3

outs ares1, ares2
endin

</CsInstruments>
<CsScore>
f 1 0 1024 10 1 ;Sine wave
f 2 0 1024 19 0.5 0.5 270 0.5 ;Grain envelope table

f0 3600

i1 0 3600

</CsScore>
</CsoundSynthesizer>

```

Here is second example of the hvs2 opcode. It uses the file *hvs2-2.csd* [examples/hvs2-2.csd].

### Example 411. Second example of the hvs2 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o hvs2.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr=48000
ksmps=100
nchnls=2

```

```

; Example by James Hearon 2008
; Edited by Andres Cabrera

ginumPointsX init      16
ginumPointsY init      16
ginumParms   init      3

;Generate 9 tables with arbitrary points
gitmp ftgen 100, 0, 16, -2, 70, 260, 390, 180, 200, 300, 980, 126, \
      330, 860, 580, 467, 220, 399, 1026, 1500
gitmp ftgen 200, 0, 16, -2, 100, 200, 300, 140, 600, 700, 880, 126, \
      330, 560, 780, 167, 220, 999, 1026, 1500
gitmp ftgen 300, 0, 16, -2, 400, 200, 300, 540, 600, 700, 880, 126, \
      330, 160, 780, 167, 820, 999, 1026, 1500
gitmp ftgen 400, 0, 16, -2, 100, 200, 800, 640, 600, 300, 880, 126, \
      330, 660, 780, 167, 220, 999, 1026, 1500
gitmp ftgen 500, 0, 16, -2, 200, 200, 360, 440, 600, 700, 880, 126, \
      330, 560, 380, 167, 220, 499, 1026, 1500
gitmp ftgen 600, 0, 16, -2, 100, 600, 300, 840, 600, 700, 880, 126, \
      330, 260, 980, 367, 120, 399, 1026, 1500
gitmp ftgen 700, 0, 16, -2, 100, 200, 300, 340, 200, 500, 380, 126, \
      330, 860, 780, 867, 120, 999, 1026, 1500
gitmp ftgen 800, 0, 16, -2, 100, 600, 300, 240, 200, 700, 880, 126, \
      130, 560, 980, 167, 220, 499, 1026, 1500
gitmp ftgen 900, 0, 16, -2, 100, 800, 200, 140, 600, 700, 680, 126, \
      330, 560, 780, 167, 120, 299, 1026, 1500

giOutTab ftgen 5,0,8, -2, 0
giPosTab ftgen 6,0,32, -2, 0,1,2,3,4,5,6,7,8,9,10, 11, 15, 14, 13, 12
giSnapTab ftgen 8,0,64, -2, 1,1,1, 2,0,0, 3,2,0, 2,2,2, \
      5,2,1, 2,3,4, 6,1,7, 0,0,0, 1,3,5, 3,4,4, 1,5,8, 1,1,5, \
      4,3,2, 3,4,5, 7,6,5, 7,8,9

tb0_init      giOutTab

      FLpanel "hsv2",440,100,10,10,0
gk1,ih1 FLslider "X", 0,1, 0, 5, -1, 400,20, 20,10
gk2, ih2 FLslider "Y", 0, 1, 0, 5, -1, 400, 20, 20, 50
      FLpanel_end

      FLpanel "hvsBox",280,280,500,1000,0
;ihandle FLhvsBox inumlinesX, inumlinesY, iwidth, iheight, ix, iy [, image]
gih1 FLhvsBox 16, 16, 250, 250, 10, 1
      FLpanel_end
      FLrun

      instr 1
FLhvsBoxSetValue gk1, gk2, gih1

hvs2      gk1,gk2, ginumParms, ginumPointsX, ginumPointsY, giOutTab, giPosTab, giSnapTab ;, iConfigTab

k0 init      0
k1 init      1
k2 init      2
kspeed init  0

kspeed = int((tb0(k2)) + 1)*.10

kenv oscil 25000, kspeed*16, 10

k1 phasor kspeed ;slow phasor: 200 sec.
kpch tableikt k1 * 16, int((tb0(k1)) +1)*100 ;scale phasor * length
a1 oscilikt kenv, kpch, int(tb0(k0)) +1000;scale pitch slightly
ahp butterlp a1, 2500
outs ahp, ahp

```

```
    endin

    </CsInstruments>
    <CsScore>

    f 10 0 1024 20 5 ;use of windowing function
    f1000 0 1024 10 .33 .25 .5
    f1001 0 1024 10 1
    f1002 0 1024 10 .5 .25 .05
    f1003 0 1024 10 .05 .10 .3 .5 1
    f1004 0 1024 10 1 .5 .25 .125 .625
    f1005 0 1024 10 .33 .44 .55 .66
    f1006 0 1024 10 1 1 1 1 1
    f1007 0 1024 10 .05 .25 .05 .25 .05 1

    f0 3600
    i1 0 3600

    </CsScore>
    </CsoundSynthesizer>
```

## See Also

*hvs1, hvs3, vphaseseg*

## Credits

Author: Gabriel Maldonado

New in version 5.06

## hvs3

**hvs3** — Allows three-dimensional Hyper Vectorial Synthesis (HVS) controlled by externally-updated k-variables.

### Description

*hvs3* allows three-dimensional Hyper Vectorial Synthesis (HVS) controlled by externally-updated k-variables.

### Syntax

```
hvs3 kx, ky, kz, inumParms, inumPointsX, inumPointsY, inumPointsZ, iOutTab, iPositionsTab, iSnapTab [,
```

### Initialization

*inumParms* - number of parameters controlled by the HVS. Each HVS snapshot is made up of *inumParms* elements.

*inumPointsX*, *inumPointsY*, *inumPointsZ* - number of points that each dimension of the HVS cube (or square in case of two-dimensional HVS; or line in case of one-dimensional HVS) is made up.

*iOutTab* - number of the table receiving the set of output-parameter instant values of the HVS. The total amount of parameters is defined by the *inumParms* argument.

*iPositionsTab* – a table filled with the individual positions of snapshots in the HVS matrix (see below for more information).

*iSnapTab* – a table filled with all the snapshots. Each snapshot is made up of a set of parameter values. The amount of elements contained in each snapshots is specified by the *inumParms* argument. The set of elements of each snapshot follows (and is adjacent) to the previous one in this table. So the total size of this table should be  $\geq$  to *inumParms* multiplied the number of snapshots you intend to store for the HVS.

*iConfigTab* – (optional) a table containing the behavior of the HVS for each parameter. If the value of *iConfigTab* is zero (default), this argument is ignored, meaning that each parameter is treated with linear interpolation by the HVS. If *iConfigTab* is different than zero, then it must refer to an existing table whose contents are in its turn referring to a particular kind of interpolation. In this table, a value of -1 indicates that corresponding parameter is leaved unchanged (ignored) by the HVS; a value of zero indicates that corresponding parameter is treated with linear-interpolation; each other values must be integer numbers indicating an existing table filled with a shape which will determine the kind of special interpolation to be used (table-based interpolation).

### Performance

*kx*, *ky*, *kz* - these are externally-modified variables which controls the motion of the pointer in the HVS matrix cube (or square or line in case of HVS matrices made up of less than 3 dimensions). The range of these input arguments must be 0 to 1.

Hyper Vectorial Synthesis is a technique that allows control of a huge set of parameters by using a simple and global approach. The key concepts of the HVS are:

The set of HVS parameters, whose amount is fixed and defined by the *inumParms* argument. During the HVS performance, all these parameters are variant and can be applied to any sound synthesis technique, as

well as to any global control for algorithmic composition and any other kind of level. The user must previously define several sets of fixed values for each HVS parameter, each set corresponding to a determinate synthesis configuration. Each set of values is called snapshot, and can be considered as the coordinates of a bound of a multi-dimensional space. The HVS consists on moving a point in this multi-dimensional space (by using a special motion pointer, see below), according and inside the bounds defined by the snapshots. You can fix any amount of HVS parameters (each parameter being a dimension of the multi-dimensional space), even a huge number, the limit only depends on the processing power (and the memory) of your computer and on the complexity of the sound-synthesis you will use.

The HVS cube (or square or line). This is the matrix (of 3, 2 or 1 dimensions, according to the `hvs` opcode you intend to use) of “mainstays” (or pivot) points of HVS. The total amount of pivot-points depends on the value of the `inumPointsX`, `inumPointsY` and `inumPointsZ` arguments. In the case of a 3-dimensional HVS matrix you can define, for instance, 3 points for the X dimension, 5 for the Y dimension and 2 for the Z dimension. In this case, the total number of pivot-points is  $3 * 5 * 2 = 30$ . With this set of pivot points, the cube is divided into smaller cubed zones each one bounded by eight nearby points. Each point is numbered. The numeral order of these points is established in the following way: number zero is the first point, number 1 the second and so on. Assuming you are using a 3-dimensional HVS cube having the number of points above mentioned (i.e. 3, 5 and 2 respectively for the X, Y and Z axis), the first point (point zero) is the upper-left-front vertex of the cube, by facing the XY plane of the cube. The second point is the middle point of the upper front edge of the cube and so on. You can refer to the figure below in order to understand how the numeral order of the pivot-points proceeds:

For the 2-dimensional HVS, it is the same, by only omitting the rear cube face, so each zone is bounded by 4 pivot-points instead of 8. For the 1-dimensional HVS, the whole thing is even simpler because it is a line with the pivot-points proceeding from left to right. Each point is coupled with a snapshot.

Snapshot order, as stored into the `iSnapTab`, can or cannot follow the order of the pivot-points numbers. In fact it is possible to alter this order by means the `iPositionsTab`, a table that remaps the position of each snapshot in relation to the pivot points. The `iPositionsTab` is made up of the positions of the snapshots (contained in the `iSnapTab`) in the two-dimensional grid. Each subsequent element is actually a pointer representing the position in the `iSnapTab`. For example, in a 2-dimensional HVS matrix such as the following (in this case having `inumPointsX = 3` and `inumPointsY = 5`):

**Table 11.**

5	7	1
3	4	9
6	2	0
4	1	3
8	2	7

These numbers (to be stored in the `iSnapTab` table by using, for instance, the GEN02 function generator) represents the snapshot position within the grid (in this case a 3x5 matrix). So, the first element 5, has index zero and represents the 6th (element zero is the first) snapshot contained in the `iSnapTab`, the second element 7 represents the 8th element of `iSnapTab` and so on. Summing up, the vertices of each zone (a cubed zone is delimited by 8 vertices; a squared zone by 4 vertices and a linear zone by 2 points) are coupled with a determinate snapshot, whose number is remapped by the `iSnapTab`.

Output values of the HVS are influenced by the motion pointer, a point whose position, in the HVS cube (or square or segment) is determined by the `kx`, `ky` and `kz` arguments. The values of these arguments, which must be in the 0 to 1 range, are externally set by the user. The output values, whose amount is equal to the `inumParms` argument, are stored in the `iOutTab`, a table that must be already allocated by the user, and must be at least `inumParms` size. In what way the motion pointer influences the output? Well, when the motion

pointer falls in a determinate cubed zone, delimited, for instance, by 8 vertices (or pivot points), we assume that each vertex has associated a different snapshot (i.e. a set of *inumParms* values), well, the output will be the weighted average value of the 8 vertices, calculated according on the distance of the motion pointer from each of the 8 vertices. In the case of a default behavior, when the *iConfigTab* argument is not set, the exact output is calculated by using linear interpolation which is applied to each different parameter of the HVS. Anyway, it is possible to influence this behavior by setting the *iConfigTab* argument to a number of a table whose contents can affect one or more HVS parameters. The *iConfigTab* table elements are associated to each HVS parameter and their values affect the HVS output in the following way:

- If *iConfigTab* is equal to -1, corresponding output is skipped, i.e. the element is not calculated, leaving corresponding element value in the *iOutTab* unchanged;
- If *iConfigTab* is equal to zero, then the normal HVS output is calculated (by using weighted average of the nearest vertex of current zone where it falls the motion pointer);
- If *iConfigTab* element is equal to an integer number > zero, then the contents of a table having that number is used as a shape of a table-based interpolation.

## See Also

*hvs1, hvs2, vphaseseg*

## Credits

Author: Gabriel Maldonado

New in version 5.06

# hypot

hypot — Euclidean distance function.

## Description

Returns the value of  $\sqrt{\text{arg1}*\text{arg1}+\text{arg2}*\text{arg2}}$ .

## Syntax

```
iRes[] hypot iArg1[], iArg2[]  
kRes[] hypot kArg1[], kArg2[]
```

## Initialization

*iArg1*[]/2 -- the operands.

## Performance

*kArg1*[]/2 -- the operands

## Examples

Here is an example of the hypot opcode. It uses the file *hypot.csd* [examples/hypot.csd].

### Example 412. Example of the hypot opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsoundOptions>  
</CsoundOptions>  
<CsoundInstruments>  
  
instr 1  
  
iArg1[] fillarray 1,2,3  
iArg2[] fillarray 4,5,6  
iRes[] hypot iArg1,iArg2  
ik init 0  
  
while ik < lenarray(iRes) do  
  print iRes[ik]  
  ik += 1  
od  
  
endin  
  
</CsoundInstruments>  
<CsoundScore>  
i1 0 0  
</CsoundScore>
```

</CsoundSynthesizer>

## Credits

Author: Victor Lazzarini  
2017

# i

*i* — Returns an init-type equivalent of a k-rate argument, or directly returns an i-rate argument.

## Description

Returns an init-type equivalent of a k-rate argument or array element or directly returns an i-rate argument.

## Syntax

`i(x)` (control-rate or init-rate arg)

`i(karray, index1, ...)` (k-array with indices)

Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.



### Note

Using `i()` with a k-rate expression argument is not recommended, and can produce unexpected results.

To get the value from an array element reliably the second form should be used.

## See Also

*a, k, abs, exp, frac, int, log, log10, sqrt*

More information on this opcode: <http://www.csoundjournal.com/issue10/CsoundRates.html> , written by Andrés Cabrera

## Credits

i-rate arg version of function new since 5.14

Array format new in version 6.08

## if

if — Branches conditionally at initialization or during performance time.

### Description

*if...igoto* -- conditional branch at initialization time, depending on the truth value of the logical expression *ia R ib*. The branch is taken only if the result is true.

*if...kgoto* -- conditional branch during performance time, depending on the truth value of the logical expression *ka R kb*. The branch is taken only if the result is true.

*if...goto* -- combination of the above. Condition tested on every pass.

*if...then* -- allows the ability to specify conditional *if/else/endif* blocks. All *if...then* blocks must end with an *endif* statement. *elseif* and *else* statements are optional. Any number of *elseif* statements are allowed. Only one *else* statement may occur and it must be the last conditional statement before the *endif* statement. Nested *if...then* blocks are allowed.



#### Note

Note that if the condition uses a k-rate variable (for instance, “if kval > 0”), the *if...goto* or *if...then* statement will be ignored during the i-time pass. This allows for opcode initialization, even if the k-rate variable has already been assigned an appropriate value by an earlier init statement.

### Syntax

```
if ia R ib igoto label
```

```
if ka R kb kgoto label
```

```
if xa R xb goto label
```

```
if xa R xb then
```

where *label* is in the same instrument block and is not an expression, and where *R* is one of the Relational operators (<, =, <=, ==, !=) (and = for convenience, see also under *Conditional Values*).

If *goto* or *then* is used instead of *kgoto* or *igoto*, the behavior is determined by the type being compared. If the comparison used k-type variables, *kgoto* is used and viceversa.



#### Note

*If/then/goto* statements cannot do audio-type comparisons. You can't put a-type variables in the comparison expressions for these opcodes. The reason for this is that audio variables are actually vectors, which can't be compared in the same way as scalars. If you need to compare individual audio samples, use *kr = 1* or *Comparators*

### Examples

Here is an example of the *if...igoto* combination. It uses the file *igoto.csd* [examples/igoto.csd].

**Example 413. Example of the if...igoto combination.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o igoto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Get the value of the 4th p-field from the score.
iparam = p4

; If iparam is 1 then play the high note.
; If not then play the low note.
if (iparam == 1) igoto highnote
    igoto lownote

highnote:
    ifreq = 880
    goto playit

lownote:
    ifreq = 440
    goto playit

playit:
; Print the values of iparam and ifreq.
print iparam
print ifreq

a1 oscil 10000, ifreq, 1
out a1
endin

</CsInstruments>
<CsScore>

; Table #1: a simple sine wave.
f 1 0 32768 10 1

; p4: 1 = high note, anything else = low note
; Play Instrument #1 for one second, a low note.
i 1 0 1 0
; Play a Instrument #1 for one second, a high note.
i 1 1 1 1
e

</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like this:

```
instr 1: iparam = 0.000
instr 1: ifreq = 440.000
instr 1: iparam = 1.000
instr 1: ifreq = 880.000
```

Here is an example of the if...kgoto combination. It uses the file *kgoto.csd* [examples/kgoto.csd].

#### Example 414. Example of the if...kgoto combination.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o kgoto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Change kval linearly from 0 to 2 over
; the period set by the third p-field.
kval line 0, p3, 2

; If kval is greater than or equal to 1 then play the high note.
; If not then play the low note.
if (kval >= 1) kgoto highnote
   kgoto lownote

highnote:
   kfreq = 880
   goto playit

lownote:
   kfreq = 440
   goto playit

playit:
; Print the values of kval and kfreq.
printks "kval = %f, kfreq = %f\\n", 1, kval, kfreq

a1 oscil 10000, kfreq, 1
out a1
endin

</CsInstruments>
<CsScore>

; Table #1: a simple sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e
```

```
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
kval = 0.000000, kfreq = 440.000000
kval = 0.999732, kfreq = 440.000000
kval = 1.999639, kfreq = 880.000000
```

## Examples

Here is an example of the if...then combo. It uses the file *ifthen.csd* [examples/ifthen.csd].

### Example 415. Example of the if...then combo.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o ifthen.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Get the note value from the fourth p-field.
knote = p4

; Does the user want a low note?
if (knote == 0) then
  kcps = 220
; Does the user want a middle note?
elseif (knote == 1) then
  kcps = 440
; Does the user want a high note?
elseif (knote == 2) then
  kcps = 880
endif

; Create the note.
kamp init 25000
ifn = 1
a1 oscili kamp, kcps, ifn

out a1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1
```

```
; p4: 0=low note, 1=middle note, 2=high note.  
; Play Instrument #1 for one second, low note.  
i 1 0 1 0  
; Play Instrument #1 for one second, middle note.  
i 1 1 1 1  
; Play Instrument #1 for one second, high note.  
i 1 2 1 2  
e
```

```
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*elseif, else, endif, goto, igoto, kgoto, tgoto, timeout*

## Credits

Examples written by Kevin Conder.

Added a note by Jim Aikin.

February 2004. Added a note by Matt Ingalls.

# fftinvs

fftinvs — Complex-to-complex Inverse Fast Fourier Transform.

## Description

Applies an Inverse Fast Fourier Transform to a complex-value input 1-dimensional array producing a complex-valued output. The output is another array containing the complex-valued signal, and both arrays are arranged in interleaved real-imaginary format. The output array will have the same size as the input, and the transform size will be equivalent to half of the length of the array. Non-power-of-two transforms are limited to even sizes with not too many factors.

## Syntax

```
kout[] fftinvs kin[]
```

## Performance

*kout[]* -- output array containing the complex-valued output. It will be created if it does not exist.

*kin[]* -- input array containing the complex-valued input.

## Examples

Here is an example of the `fftinvs` opcode. It uses the file *ifft.csd* [examples/ifft.csd].

### Example 416. Example of the `fftinvs` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-d -o dac
</CsOptions>
<CsInstruments>
ksmps = 64

instr 1
  ifftsize = 1024
  kcnt init 0
  kIn[] init ifftsize
  kOut[] init ifftsize

  a1 oscili 0dbfs/2, 440

  if kcnt >= ifftsize then
    kCmplx[] r2c kIn
    kSpec[] fft kCmplx
    kCmplx ifftinvs kSpec
    kOut c2r kCmplx
    kcnt = 0
  endif

  kIn[] shiftin a1
```

```
a2 shiftout kOut  
kcnt += ksmps  
  out a2  
endin  
</CsInstruments>  
<CsScore>  
i 0 10  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

Vectorial opcodes, array opcodes

## Credits

Author: Victor Lazzarini  
NUI Maynooth  
2014

New in version 6.04

# igoto

igoto — Transfer control during the i-time pass.

## Description

During the i-time pass only, unconditionally transfer control to the statement labeled by *label*.

## Syntax

```
igoto label
```

where *label* is in the same instrument block and is not an expression.

## Examples

Here is an example of the igoto opcode. It uses the file *igoto.csd* [examples/igoto.csd].

### Example 417. Example of the igoto opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o igoto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Get the value of the 4th p-field from the score.
iparam = p4

; If iparam is 1 then play the high note.
; If not then play the low note.
if (iparam == 1) igoto highnote
    igoto lownote

highnote:
    ifreq = 880
    goto playit

lownote:
    ifreq = 440
    goto playit
```

```
playit:
  ; Print the values of iparam and ifreq.
  print iparam
  print ifreq

  a1 oscil 10000, ifreq, 1
  out a1
endin

</CsInstruments>
<CsScore>

; Table #1: a simple sine wave.
f 1 0 32768 10 1

; p4: 1 = high note, anything else = low note
; Play Instrument #1 for one second, a low note.
i 1 0 1 0
; Play a Instrument #1 for one second, a high note.
i 1 1 1 1
e

</CsScore>
</CsSoundSynthesizer>
```

Its output should include lines like this:

```
instr 1: iparam = 0.000
instr 1: ifreq = 440.000
instr 1: iparam = 1.000
instr 1: ifreq = 880.000
```

## See Also

*cggoto, cigoto, ckgoto, goto, if, kgoto, rigoto, tigoto, timeout*

## Credits

Example written by Kevin Conder.

# ihold

ihold — Creates a held note.

## Description

Causes a finite-duration note to become a “held” note

## Syntax

ihold

## Performance

*ihold* -- this i-time statement causes a finite-duration note to become a “held” note. It thus has the same effect as a negative p3 ( see score *i Statement*), except that p3 here remains positive and the instrument reclassifies itself to being held indefinitely. The note can be turned off explicitly with *turnoff*, or its space taken over by another note of the same instrument number (i.e. it is tied into that note). Effective at i-time only; no-op during a *reinit* pass.

## Examples

Here is an example of the *ihold* opcode. It uses the file *ihold.csd* [examples/ihold.csd].

### Example 418. Example of the *ihold* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o ihold.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; A simple oscillator with its note held indefinitely.
a1 oscil 10000, 440, 1
ihold

; If p4 equals 0, turn the note off.
if (p4 == 0) kgoto offnow
kgoto playit
```

```
offnow:
  ; Turn the note off now.
  turnoff

playit:
  ; Play the note.
  out a1
endin

</CsInstruments>
<CsScore>

; Table #1: an ordinary sine wave.
f 1 0 32768 10 1

; p4 = turn the note off (if it is equal to 0).
; Start playing Instrument #1.
i 1 0 1 1
; Turn Instrument #1 off after 3 seconds.
i 1 3 1 0
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*i* Statement, *turnoff*

## Credits

Example written by Kevin Conder.

# imagecreate

imagecreate — Create an empty image of a given size.

## Description

Create an empty image of a given size. Individual pixel values can then be set with *imagegetpixel*.

## Syntax

```
iimagenum imagecreate iwidth, iheight
```

## Initialization

*iimagenum* -- number assigned to the created image.

*iwidth* -- desired image width.

*iheight* -- desired image height.

## Examples

Here is an example of the imagecreate opcode. It uses the file *imageopcodes.csd* [examples/imageopcodes.csd].

### Example 419. Example of the imageload opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n ;no sound output
</CsOptions>
<CsInstruments>

sr=48000
ksmps=1
nchnls=2

; this test .csd copies image.png into a new file 'imageout.png'

giimage1 imageload "image.png"
giimagew, giimageh imagesize giimage1
giimage2 imagecreate giimagew,giimageh

instr 1

kndx = 0
kx_ linseg 0, p3, 1
;print imagewidth and imageheigth of image.png
prints "imagewidth = %f pixels, imageheigth = %f pixels\\n", giimagew, giimageh

myloop:
ky_ = kndx/(giimageh)
kr_, kg_, kb_ imagegetpixel giimage1, kx_, ky_
imagesetpixel giimage2, kx_, ky_, kr_, kg_, kb_
```

```
loop_lt kndx, 0.5, giimageh, myloop
  endin

  instr 2

imagesave giimage2, "imageout.png"
  endin

  instr 3
imagefree giimage1
imagefree giimage2
  endin

</CsInstruments>
<CsScore>

i1 1 1
i2 2 1
i3 3 1
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*imageload, imagesize, imagesave, imagegetpixel, imagesetpixel, imagefree*

## Credits

Author: Cesare Marilungo

New in version 5.08

# imagefree

imagecreate — Frees memory allocated for a previously loaded or created image.

## Description

Frees memory allocated for a previously loaded or created image.

## Syntax

```
imagefree iimagenum
```

## Initialization

*iimagenum* -- reference of the image to free.

## Examples

Here is an example of the imagefree opcode. It uses the file *imageopcodes.csd* [examples/imageopcodes.csd].

### Example 420. Example of the imagefree opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n ;no sound output
</CsOptions>
<CsInstruments>

sr=48000
ksmps=1
nchnls=2

; this test .csd copies image.png into a new file 'imageout.png'

giimage1 imageload "image.png"
giimagew, giimageh imagesize giimage1
giimage2 imagecreate giimagew,giimageh

instr 1

kndx = 0
kx_ linseg 0, p3, 1
;print imagewidth and imageheigth of image.png
prints "imagewidth = %f pixels, imageheigth = %f pixels\\n", giimagew, giimageh

myloop:
ky_ = kndx/(giimageh)
kr_, kg_, kb_ imagegetpixel giimage1, kx_, ky_
imagesetpixel giimage2, kx_, ky_, kr_, kg_, kb_
loop_lt kndx, 0.5, giimageh, myloop
endin

instr 2
```

```
imagesave giimage2, "imageout.png"  
  endin  
  
  instr 3  
imagefree giimage1  
imagefree giimage2  
  endin  
  
</CsInstruments>  
<CsScore>  
  
i1 1 1  
i2 2 1  
i3 3 1  
e  
  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*imageload, imagecreate, imagesize, imagesave, imagegetpixel, imagesetpixel*

## Credits

Author: Cesare Marilungo

New in version 5.08

# imagegetpixel

imagegetpixel — Return the RGB pixel values of a previously opened or created image.

## Description

Return the RGB pixel values of a previously opened or created image. An image can be loaded with *imageload*. An empty image can be created with *imagecreate*.

## Syntax

```
ared, agreen, ablue imagegetpixel iimagenum, ax, ay
kred, kgreen, kblue imagegetpixel iimagenum, kx, ky
```

## Initialization

*iimagenum* -- the reference of the image.. It should be a value returned by *imageload* or *imagecreate*.

## Performance

*ax (kx)* -- horizontal pixel position (must be a float from 0 to 1).

*ay (ky)* -- vertical pixel position (must be a float from 0 to 1).

*ared (kred)* -- red value of the pixel (mapped to a float from 0 to 1).

*agreen (kgreen)* -- green value of the pixel (mapped to a float from 0 to 1).

*ablue (kblue)* -- blue value of the pixel (mapped to a float from 0 to 1).

## Examples

Here is an example of the imagegetpixel opcode. It uses the files *imageopcodesdemo2.csd* [examples/imageopcodesdemo2.csd] *test1.png* [examples/test1.png] and *test2.png* [examples/test2.png].

### Example 421. Example of the imagegetpixel opcode.

```
<CsoundSynthesizer>

<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc    ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o imageopcodesdemo2.wav -W ;; for file output any platform
</CsOptions>

<CsInstruments>
sr          =      48000
ksmps       =      100
nchnls     =      2
```

```
;By Cesare Marilungo 2008
zakinit 10,1

;Load the image - should be 512x512 pixels
giimage imageload "test1.png"
;giimage imageload "test2.png" ;--try this too
giimagew, giimageh imagesize giimage

giwave ftgen 1, 0, 1024, 10, 1
gifrqs ftgen 2,0,512,-5, 1,512,10
giamps ftgen 3, 0, 512, 10, 1

  instr 100

kindex = 0
icnt = giimageh
kx_ linseg 0, p3, 1
kenv linseg 0, .2, 500, p3 - .4, 500, .2, 0

;Read a column of pixels and store the red values
;inside the table 'giamps'
loop:
  ky_ = kindex/giimageh

  ;Get the pixel color values at kx_, ky_
  kred, kgreen, kblue imagegetpixel giimage, kx_, ky_

  ;Write the red values inside the table 'giamps'
  tablew kred, kindex, giamps
  kindex = kindex+1

if (kindex < icnt) kgoto loop

;Use an oscillator bank (additive synthesis) to generate sound
;setting amplitudes for each partial according to the image
asig adsynt kenv, 220, giwave, gifrqs, giamps, icnt, 2
outs asig, asig

  endin

  instr 101
  ;Free memory used by the image
  imagefree giimage
  endin

</CsInstruments>
<CsScore>

t 0 60

i100 1 20
i101 21 1

e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*imageload, imagecreate, imagesize, imagesave, imagesetpixel, imagefree*

## Credits

Author: Cesare Marilungo

New in version 5.08

# imageload

imageload — Load an image.

## Description

Load an image and return a reference to it. Individual pixel values can then be accessed with *imagegetpixel*.



### Note

The image processing opcodes can only load png images

## Syntax

```
iimagenum imageload filename
```

## Initialization

*iimagenum* -- number assigned to the loaded image.

*filename* -- The filename of the image to load (should be a valid PNG image file).

## Examples

Here is an example of the imageload opcode. It uses the file *imageopcodes.csd* [examples/imageopcodes.csd].

### Example 422. Example of the imageload opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n ;no sound output
</CsOptions>
<CsInstruments>

sr=48000
ksmps=1
nchnls=2

; this test .csd copies image.png into a new file 'imageout.png'

giimage1 imageload "image.png"
giimagew, giimageh imagesize giimage1
giimage2 imagecreate giimagew,giimageh

instr 1

kndx = 0
kx_ linseg 0, p3, 1
;print imagewidth and imageheight of image.png
prints "imagewidth = %f pixels, imageheight = %f pixels\\n", giimagew, giimageh

myloop:
```

```
ky_ = kndx/(giimageh)
kr_, kg_, kb_ imagegetpixel giimage1, kx_, ky_
imagesetpixel giimage2, kx_, ky_, kr_, kg_, kb_
loop_lt kndx, 0.5, giimageh, myloop
  endin

  instr 2

imagesave giimage2, "imageout.png"
endin

  instr 3
imagefree giimage1
imagefree giimage2
endin

</CsInstruments>
<CsScore>

i1 1 1
i2 2 1
i3 3 1
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*imagecreate, imagesize, imagesave, imagegetpixel, imagesetpixel, imagefree*

## Credits

Author: Cesare Marilungo

New in version 5.08

# imagesave

imagesave — Save a previously created image.

## Description

Save a previously created image. An empty image can be created with *imagecreate* and its pixel RGB values can be set with *imagesetpixel*. The image will be saved in PNG format.

## Syntax

```
imagesave iimagenum, filename
```

## Initialization

*iimagenum* -- the reference of the image to be save. It should be a value returned by *imagecreate*.

*filename* -- The filename to use to save the image.

## Examples

Here is an example of the imagesave opcode. It uses the file *imageopcodes.csd* [examples/imageopcodes.csd].

### Example 423. Example of the imagesave opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n ;no sound output
</CsOptions>
<CsInstruments>

sr=48000
ksmps=1
nchnls=2

; this test .csd copies image.png into a new file 'imageout.png'

giimage1 imageload "image.png"
giimagew, giimageh imagesize giimage1
giimage2 imagecreate giimagew,giimageh

instr 1

kndx = 0
kx_ linseg 0, p3, 1
;print imagewidth and imageheigth of image.png
prints "imagewidth = %f pixels, imageheigth = %f pixels\\n", giimagew, giimageh

myloop:
ky_ = kndx/(giimageh)
kr_, kg_, kb_ imagegetpixel giimage1, kx_, ky_
imagesetpixel giimage2, kx_, ky_, kr_, kg_, kb_
loop_lt kndx, 0.5, giimageh, myloop
```

```
    endin

    instr 2
    imagesave giimage2, "imageout.png"
    endin

    instr 3
    imagefree giimage1
    imagefree giimage2
    endin

</CsInstruments>
<CsScore>

i1 1 1
i2 2 1
i3 3 1
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*imageload, imagecreate, imagesize, imagegetpixel, imagesetpixel, imagefree*

## Credits

Author: Cesare Marilungo

New in version 5.08

# imagesetpixel

`imagesetpixel` — Set the RGB value of a pixel inside a previously opened or created image.

## Description

Set the RGB value of a pixel inside a previously opened or created image. An image can be loaded with *imageload*. An empty image can be created with *imagecreate* and saved with *imagesave*.

## Syntax

```
imagesetpixel iimagenum, ax, ay, ared, agreen, ablue
```

```
imagesetpixel iimagenum, kx, ky, kred, kgreen, kblue
```

## Initialization

*iimagenum* -- the reference of the image.. It should be a value returned by *imageload* or *imagecreate*.

## Performance

*ax* (*kx*) -- horizontal pixel position (must be a float from 0 to 1).

*ay* (*ky*) -- vertical pixel position (must be a float from 0 to 1).

*ared* (*kred*) -- red value of the pixel (mapped to a float from 0 to 1).

*agreen* (*kgreen*) -- green value of the pixel (mapped to a float from 0 to 1).

*ablue* (*kblue*) -- blue value of the pixel (mapped to a float from 0 to 1).

## Examples

Here is an example of the `imagesetpixel` opcode. It uses the file *imageopcodes.csd* [examples/imageopcodes.csd].

### Example 424. Example of the `imagesetpixel` opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n ;no sound output
</CsOptions>
<CsInstruments>

sr=48000
ksmps=1
nchnls=2

; this test .csd copies image.png into a new file 'imageout.png'

giimage1 imageload "image.png"
giimagew, giimageh imagesize giimage1
```

```
giimage2 imagecreate giimagew,giimageh

    instr 1

kndx = 0
kx_ linseg 0, p3, 1
;print imagewidth and imageheigth of image.png
prints "imagewidth = %f pixels, imageheigth = %f pixels\\n", giimagew, giimageh

myloop:
ky_ = kndx/(giimageh)
kr_, kg_, kb_ imagegetpixel giimage1, kx_, ky_
imagesetpixel giimage2, kx_, ky_, kr_, kg_, kb_
loop_lt kndx, 0.5, giimageh, myloop
    endin

    instr 2

imagesave giimage2, "imageout.png"
    endin

    instr 3
imagefree giimage1
imagefree giimage2
    endin

</CsInstruments>
<CsScore>

i1 1 1
i2 2 1
i3 3 1
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*imagerload, imagecreate, imagesize, imagesave, imagegetpixel, imagefree*

## Credits

Author: Cesare Marilungo

New in version 5.08

# imagesize

`imagesize` — Return the width and height of a previously opened or created image.

## Description

Return the width and height of a previously opened or created image. An image can be loaded with *imload*. An empty image can be created with *imagecreate*.

## Syntax

```
iwidth, iheight imagesize iimagenum
```

## Initialization

*iimagenum* -- the reference of the image.. It should be a value returned by *imload* or *imagecreate*.

*iwidth* -- image width.

*iheight* -- image height.

## Examples

Here is an example of the `imagesize` opcode. It uses the file *imageopcodes.csd* [examples/imageopcodes.csd].

### Example 425. Example of the `imagesize` opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n ;no sound output
</CsOptions>
<CsInstruments>

sr=48000
ksmps=1
nchnls=2

; this test .csd copies image.png into a new file 'imageout.png'

giimage1 imload "image.png"
giimagew, giimageh imagesize giimage1
giimage2 imagecreate giimagew,giimageh

instr 1

kndx = 0
kx_ linseg 0, p3, 1
;print imagewidth and imageheight of image.png
prints "imagewidth = %f pixels, imageheight = %f pixels\\n", giimagew, giimageh

myloop:
ky_ = kndx/(giimageh)
kr_, kg_, kb_ imagegetpixel giimage1, kx_, ky_
```

```
imagesetpixel giimage2, kx_, ky_, kr_, kg_, kb_  
loop_lt kndx, 0.5, giimageh, myloop  
  endin  
  
  instr 2  
  
imagesave giimage2, "imageout.png"  
  endin  
  
  instr 3  
imagefree giimage1  
imagefree giimage2  
  endin  
  
</CsInstruments>  
<CsScore>  
  
i1 1 1  
i2 2 1  
i3 3 1  
e  
  
</CsScore>  
</CsSoundSynthesizer>
```

Its output should include a line like this:

```
imagewidth = 180.000000 pixels, imageheight = 135.000000 pixels
```

## See Also

*imageload, imagecreate, imagesave, imagegetpixel, imagesetpixel, imagefree*

## Credits

Author: Cesare Marilungo

New in version 5.08

# in

in — Reads mono audio data from an external device or stream.

## Description

Reads audio data from an external device or stream.



### Warning

There are two versions of this opcode. The first is designed to be used only with orchestras that have `inchnls=1`. Doing so with orchestras with `inchnls > 1` will cause incorrect audio input.

The second form will read multiple channels into an array.

## Syntax

```
arl in
```

```
aarray in
```

## Performance

Form 1 reads mono audio data from an external device or stream. If the command-line `-i` flag is set, sound is read continuously from the audio input stream (e.g. `stdin` or a soundfile) into an internal buffer. Any number of these opcodes can read freely from this buffer.

The second format will read upto `inchnls` of audio into an audio array, which must be initialised.

## Examples

Here is an example of the `in` opcode. It uses the file `in.csd` [examples/in.csd].

### Example 426. Example of the `in` opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -idac ;;;realtime audio I/O
; For Non-realtime ouput leave only the line below:
; in.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;1 channel in, two channels out
ainl in ;grab your mic and sing
```

```
adel linseg 0, p3*.5, 0.02, p3*.5, 0 ;max delay time = 20ms
aout flanger ain1, adel, .7
      fout "in_1.wav", 14, aout, aout ;write to stereo file,
      outs aout, aout ;16 bits with header

endin
</CsInstruments>
<CsScore>

i 1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*diskin, inh, inh, ino, inq, ins, soundin*

## Credits

Authors: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997  
Authors: John ffitch  
NUIM, 2013

Already in version 3.30

Array version in versio 6.01

## in32

`in32` — Reads a 32-channel audio signal from an external device or stream.

### Description

Reads a 32-channel audio signal from an external device or stream.



#### Warning

This opcode is designed to be used only with orchestras that have `nchnls_i=32`. Doing so with orchestras with `nchnls_i > 32` will cause incorrect audio input.

### Syntax

```
ar1, ar2, ar3, ar4, ar5, ar6, ar7, ar8, ar9, ar10, ar11, ar12, ar13, ar14, \  
ar15, ar16, ar17, ar18, ar19, ar20, ar21, ar22, ar23, ar24, ar25, ar26, \  
ar27, ar28, ar29, ar30, ar31, ar32 in32
```

### Performance

`in32` reads a 32-channel audio signal from an external device or stream. If the command-line `-i` flag is set, sound is read continuously from the audio input stream (e.g. `stdin` or a soundfile) into an internal buffer.

### See Also

`inch`, `inx`, `inz`

### Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2000

New in Csound Version 4.07

# inch

inch — Reads from numbered channels in an external audio signal or stream.

## Description

Reads from numbered channels in an external audio signal or stream.

## Syntax

```
ain1[, ...] inch kchan1[,...]
```

## Performance

*ain1*, ... - input audio signals

*kchan1*,... - channel numbers

*inch* reads from numbered channels determined by the corresponding *kchan* into the associated *ain*. If the command-line *-i* flag is set, sound is read continuously from the audio input stream (e.g. *stdin* or a soundfile). *inch* can also be used to receive audio in realtime from the audio interface using *-iadc*.



### Note

The highest number for *kchan* available for use with *inch* depends on *nchnls\_i*. If *kchan* is greater than *nchnls\_i*, *ain* will be silent. Note that *inch* will give a warning but not an error in this case.

## Examples

Here is an example of the *inch* opcode. It uses the file *inch.csd* [examples/inch.csd].

### Example 427. Example of the *inch* opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -iadc ;;realtime audio I/O
; For Non-realtime ouput leave only the line below:
; inch.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;nchnls channels in, two channels out

ain1, ainr inch 1, 2 ;grab your mic and sing
adel linseg 0, p3*.5, 0.02, p3*.5, 0 ;max delay time = 20ms
aout1 flanger ain1, adel, .7
```

```
aoutr flanger ainl, adel*2, .8
      fout "in_ch.wav", 14, aoutl, aoutr ;write to stereo file,
      outs aoutl, aoutr ;16 bits with header

endin
</CsInstruments>
<CsScore>

i 1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*in32, inx, inz*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2000

New in Csound Version 4.07

Multiple arguments from version 5.13

# inh

inh — Reads six-channel audio data from an external device or stream.

## Description

Reads six-channel audio data from an external device or stream.



### Warning

This opcode is designed to be used only with orchestras that have `nchnls_i=6`. Doing so with orchestras with `nchnls_i > 6` will cause incorrect audio input.

## Syntax

`ar1, ar2, ar3, ar4, ar5, ar6 inh`

## Performance

Reads six-channel audio data from an external device or stream. If the command-line `-i` flag is set, sound is read continuously from the audio input stream (e.g. `stdin` or a soundfile) into an internal buffer. Any number of these opcodes can read freely from this buffer.

## See Also

*diskin, in, ino, inq, ins, soundin*

## Credits

Author: John ffitc

# init

`init` — Puts the value of the i-time expression into a k-, a-rate or t- variable.

## Syntax

```
ares init iarg
ires init iarg
kres init iarg
ares, ... init iarg, ...
ires, ... init iarg, ...
kres, ... init iarg, ...
tab init isize[, ival]
```

## Description

Put the value of the i-time expression into a k- or a-rate variable.

## Initialization

Puts the value of the i-time expression *iarg* into a k-, a-rate or t- variable, i.e., initialize the result. Note that `init` provides the only case of an init-time statement being permitted to write into a perf-time (k- or a-rate) result cell; the statement has no effect at perf-time.

Since version 5.13 it is possible to initialise upto 24 variables of the same class in one statement. If there are more output variables than input expressions then the last one is repeated. It is an error to have more inputs than outputs.

The t-variable form was introduced in 5.14 and allocated space for a vector of the given size, initialised to the given value (default value is zero).

## Examples

Here is an example of the `init` opcode. It uses the file *init.csd* [examples/init.csd].

### Example 428. Example of the `init` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n ;no sound output
</CsOptions>
<CsInstruments>

sr      = 44100
ksmps  = 32
```

```

nchnls = 2
odbfs = 1

instr 1 ;shows what init does
    kinit init 0
    kinit = kinit + 1
    printk .1, kinit
endin

instr 2 ;shows what an assignment does
    knoinit = 0
    knoinit = knoinit + 1
    printk .1, knoinit
endin
</CsInstruments>
<CsScore>
;play one second each
i1 0 1
i2 2 1
e
</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like these:

```

i 1 time 0.00073: 1.00000
i 1 time 0.10014: 138.00000
i 1 time 0.20027: 276.00000
i 1 time 0.30041: 414.00000
i 1 time 0.40054: 552.00000
i 1 time 0.50068: 690.00000
i 1 time 0.60009: 827.00000
i 1 time 0.70023: 965.00000
i 1 time 0.80036: 1103.00000
i 1 time 0.90050: 1241.00000

i 2 time 2.00054: 1.00000
i 2 time 2.09995: 1.00000
i 2 time 2.20009: 1.00000
i 2 time 2.30023: 1.00000
i 2 time 2.40036: 1.00000
i 2 time 2.50050: 1.00000
i 2 time 2.59991: 1.00000
i 2 time 2.70005: 1.00000
i 2 time 2.80018: 1.00000
i 2 time 2.90032: 1.00000

```

## See Also

*=, divz, tival*

Other information on this opcode in the Floss Manuals: [http://www.flossmanuals.net/csound/ch016\\_a-initialization-and-performance-pass](http://www.flossmanuals.net/csound/ch016_a-initialization-and-performance-pass)

## Credits

Init first appeared in the original Csound, but the extension to multiple values is by

Author: John ffitth  
 University of Bath, and Codemist Ltd.  
 Bath, UK

February 2010

Multiple form new in version 5.13; t-variable form new in 5.14.

# initc14

*initc14* — Initializes the controllers used to create a 14-bit MIDI value.

## Description

Initializes the controllers used to create a 14-bit MIDI value.

## Syntax

```
initc14 ichan, ictlno1, ictlno2, ivalue
```

## Initialization

*ichan* -- MIDI channel (1-16)

*ictlno1* -- most significant byte controller number (0-127)

*ictlno2* -- least significant byte controller number (0-127)

*ivalue* -- floating point value (must be within 0 to 1)

## Performance

*initc14* can be used together with both *midic14* and *ctrl14* opcodes for initializing the first controller's value. *ivalue* argument must be set with a number within 0 to 1. An error occurs if it is not. Use the following formula to set *ivalue* according with *midic14* and *ctrl14* min and max range:

$$\text{ivalue} = (\text{initial\_value} - \text{min}) / (\text{max} - \text{min})$$

## See Also

*ctrl7*, *ctrl14*, *ctrl21*, *ctrlinit*, *initc7*, *initc21*, *midic7*, *midic14*, *midic21*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# initc21

`initc21` — Initializes the controllers used to create a 21-bit MIDI value.

## Description

Initializes the controllers used to create a 21-bit MIDI value.

## Syntax

```
initc21 ichan, ictlno1, ictlno2, ictlno3, ivalue
```

## Initialization

*ichan* -- MIDI channel (1-16)

*ictlno1* -- most significant byte controller number (0-127)

*ictlno2* -- medium significant byte controller number (0-127)

*ictlno3* -- least significant byte controller number (0-127)

*ivalue* -- floating point value (must be within 0 to 1)

## Performance

`initc21` can be used together with both `midic21` and `ctrl21` opcodes for initializing the first controller's value. *ivalue* argument must be set with a number within 0 to 1. An error occurs if it is not. Use the following formula to set *ivalue* according with `midic21` and `ctrl21` min and max range:

$$\text{ivalue} = (\text{initial\_value} - \text{min}) / (\text{max} - \text{min})$$

## See Also

`ctrl7`, `ctrl14`, `ctrl21`, `ctrlinit`, `initc7`, `initc14`, `midic7`, `midic14`, `midic21`

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# initc7

*initc7* — Initializes the controller used to create a 7-bit MIDI value.

## Description

Initializes MIDI controller *ictlno* with *ivalue*

## Syntax

```
initc7 ichan, ictlno, ivalue
```

## Initialization

*ichan* -- MIDI channel (1-16)

*ictlno* -- controller number (0-127)

*ivalue* -- floating point value (must be within 0 to 1)

## Performance

*initc7* can be used together with both *midic7* and *ctrl7* opcodes for initializing the first controller's value. *ivalue* argument must be set with a number within 0 to 1. An error occurs if it is not. Use the following formula to set *ivalue* according with *midic7* and *ctrl7* min and max range:

$$\text{ivalue} = (\text{initial\_value} - \text{min}) / (\text{max} - \text{min})$$

## Examples

Here is an example of the *initc7* opcode. It uses the file *initc7.csd* [examples/initc7.csd].

### Example 429. Example of the *initc7* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac  -M0  ;;realtime audio I/O with MIDI in
;-iadc  ;;uncomment -iadc if RT audio input is needed too
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; expects MIDI controller input on channel 1
```

```
; run and move your midi controller to see result

imax = 1
imin = 0
ichan = 1
ictlno = 7

initc7 1, 7, 1 ; start at max. volume
kamp ctrl7 ichan, ictlno, imin, imax ; controller 7
printk2 kamp
asig oscil kamp, 220, 1
outs asig, asig

endin

</CsInstruments>
<CsScore>
f 1 0 4096 10 1

i1 0 20

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*ctrl7, ctrl14, ctrl21, ctrlinit, initc14, initc21, midic7, midic14, midic21*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# inleta

inleta — Receives an arate signal into an instrument through a named port.

## Description

Receives an arate signal into an instrument through a named port.

## Syntax

```
asignal inleta Sname
```

## Initialization

*Sname* -- String name of the inlet port. The name of the inlet is implicitly qualified by the instrument name or number, so it is valid to use the same inlet name in more than one instrument (but not to use the same inlet name twice in one instrument).

## Performance

*asignal* -- audio input signal

During performance, the arate inlet signal is received from each instance of an instrument containing an outlet port to which this inlet has been connected using the connect opcode. The signals of all the outlets connected to an inlet are summed in the inlet.

## Examples

Here is an example of the inleta opcode. It uses the file *inleta.csd* [examples/inleta.csd].

### Example 430. Example of the inleta opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o inleta.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

; Connect up instruments and effects to create the signal flow graph.

connect "SimpleSine", "leftout", "Reverberator", "leftin"
connect "SimpleSine", "rightout", "Reverberator", "rightin"

connect "Moogy", "leftout", "Reverberator", "leftin"
connect "Moogy", "rightout", "Reverberator", "rightin"

connect "Reverberator", "leftout", "Compressor", "leftin"
connect "Reverberator", "rightout", "Compressor", "rightin"
```

```

connect "Compressor", "leftout", "Soundfile", "leftin"
connect "Compressor", "rightout", "Soundfile", "rightin"

; Turn on the "effect" units in the signal flow graph.

alwayson "Reverberator", 0.91, 12000
alwayson "Compressor"
alwayson "Soundfile"

; Define instruments and effects in order of signal flow.

instr SimpleSine
////////////////////////////////////
; Default values: p1 p2 p3 p4 p5 p6 p7 p8 p9 p10
pset 0, 0, 10, 0, 0, 0, 0, 0.5
iattack = 0.015
idecay = 0.07
isustain = p3
irelease = 0.3
p3 = iattack + idecay + isustain + irelease
adamping linsegr 0.0, iattack, 1.0, idecay + isustain, 1.0, irelease, 0.0
iHz = cpsmidinn(p4)
; Rescale MIDI velocity range to a musically usable range of dB.
iamplitude = ampdb(p5 / 127 * 15.0 + 60.0)
; Use ftgenonce instead of ftgen, ftgentmp, or f statement.
icosine ftgenonce 0, 0, 65537, 11, 1
aoscili oscili iamplitude, iHz, icosine
aadsr madsr iattack, idecay, 0.6, irelease
asignal = aoscili * aadsr
aleft, aright pan2 asignal, p7
; Stereo audio output to be routed in the orchestra header.
outleta "leftout", aleft
outleta "rightout", aright
endin

instr Moogy
////////////////////////////////////
; Default values: p1 p2 p3 p4 p5 p6 p7 p8 p9 p10
pset 0, 0, 10, 0, 0, 0, 0.5
iattack = 0.003
isustain = p3
irelease = 0.05
p3 = iattack + isustain + irelease
adamping linsegr 0.0, iattack, 1.0, isustain, 1.0, irelease, 0.0
iHz = cpsmidinn(p4)
; Rescale MIDI velocity range to a musically usable range of dB.
iamplitude = ampdb(p5 / 127 * 20.0 + 60.0)
print iHz, iamplitude
; Use ftgenonce instead of ftgen, ftgentmp, or f statement.
isine ftgenonce 0, 0, 65537, 10, 1
asignal vco iamplitude, iHz, 1, 0.5, isine
kfco line 2000, p3, 200
krez = 0.8
asignal moogvcf asignal, kfco, krez, 100000
asignal = asignal * adamping
aleft, aright pan2 asignal, p7
; Stereo audio output to be routed in the orchestra header.
outleta "leftout", aleft
outleta "rightout", aright
endin

instr Reverberator
////////////////////////////////////
; Stereo input.
aleftin inleta "leftin"
arightin inleta "rightin"

```

```

idelay = p4
icutoff = p5
aleft, aright reverbsc aleftin, arightin, idelay, icutoff
; Stereo output.
outleta "leftout", aleft
outleta "rightout", aright
endin

instr Compressor
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Stereo input.
aleftin inleta "leftin"
arightin inleta "rightin"
kthreshold = 25000
icompl = 0.5
icomp2 = 0.763
irtime = 0.1
ifetime = 0.1
aleftout dam aleftin, kthreshold, icomp1, icomp2, irtime, iftime
arightout dam arightin, kthreshold, icomp1, icomp2, irtime, iftime
; Stereo output.
outleta "leftout", aleftout
outleta "rightout", arightout
endin

instr Soundfile
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Stereo input.
aleftin inleta "leftin"
arightin inleta "rightin"
outs aleftin, arightin
endin

</CsInstruments>
<CsScore>

; It is not necessary to activate "effects" or create f-tables in the score!
; Overlapping notes create new instances of instruments with proper connections.

i "SimpleSine" 1 5 60 85
i "SimpleSine" 2 5 64 80
i "Moogy" 3 5 67 75
i "Moogy" 4 5 71 70
;1 extra second after the performance
e 1

</CsScore>
</CsoundSynthesizer>

```

## See Also

*outleta outletk outletkid outletf inletk inletkid inletf connect alwayson ftgenonce*

More information on this opcode: <http://www.csoundjournal.com/issue13/signalFlowGraphOpcodes.html>  
, written by Michael Gogins

## Credits

By: Michael Gogins 2009

# inletk

inletk — Receives a krate signal into an instrument from a named port.

## Description

Receives a krate signal into an instrument from a named port.

## Syntax

```
ksignal inletk Sname
```

## Initialization

*Sname* -- String name of the inlet port. The name of the inlet is implicitly qualified by the instrument name or number, so it is valid to use the same inlet name in more than one instrument (but not to use the same inlet name twice in one instrument).

## Performance

*ksignal* -- krate input signal

During performance, the krate inlet signal is received from each instance of an instrument containing an outlet port to which this inlet has been connected using the connect opcode. The signals of all the outlets connected to an inlet are summed in the inlet.

## Examples

Here is an example of the inletk opcode. It uses the file *inletk.csd* [examples/inletk.csd].

### Example 431. Example of the inletk opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o inletk.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

connect "bend", "bendout", "guitar", "bendin"

instr bend

kbend line p4, p3, p5
      outletk "bendout", kbend
```

```
    endin

    instr guitar

    kbend inletk "bendin"
    kpch pow 2, kbend/12
        printk2 kpch
    asig oscili .4, 440*kpch, 1
        outs asig, asig
    endin

</CsInstruments>
<CsScore>
f1 0 1024 10 1

i"guitar" 0 5 8.00
i"bend" 3 .2 -12 12
i"bend" 4 .1 -17 40
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*outleta outletk outletkid outletf inleta inletf connect alwayson ftgenonce*

## Credits

By: Michael Gogins 2009

# inletkid

inletkid — Receives a krate signal into an instrument from a named port.

## Description

Receives a krate signal into an instrument from a named port.

## Syntax

```
ksignal inletkid Sname, SinstanceID
```

## Initialization

*Sname* -- String name of the inlet port. The name of the inlet is implicitly qualified by the instrument name or number, so it is valid to use the same inlet name in more than one instrument (but not to use the same inlet name twice in one instrument).

*SinstanceID* -- String name of the outlet port's instance ID. This enables the inlet to discriminate between different instances of the outlet, e.g. one instance of the outlet might be created by a note specifying one instance ID, and another instance might be created by a note specifying another ID. This might be used, e.g., to situate difference instances of an instrument at different points in an Ambisonic space in a spatializing effects processor.

## Performance

*ksignal* -- krate input signal

During performance, the krate inlet signal is received from each instance of an instrument containing an outlet port to which this inlet has been connected using the connect opcode. The signals of all the outlets that are connected to an inlet, but only those share the specified instance ID, are summed in the inlet.

## See Also

*outleta outletk outletkid outletf inleta inletf connect alwayson ftgenonce*

## Credits

By: Michael Gogins 2009

# inletf

inletf — Receives an frate signal (fsig) into an instrument from a named port.

## Description

Receives an frate signal (fsig) into an instrument from a named port.

## Syntax

```
fsignal inletf Sname
```

## Initialization

*Sname* -- String name of the inlet port. The name of the inlet is implicitly qualified by the instrument name or number, so it is valid to use the same inlet name in more than one instrument (but not to use the same inlet name twice in one instrument).

## Performance

*ksignal* -- frate input signal

During performance, the frate inlet signal is received from each instance of an instrument containing an outlet port to which this inlet has been connected using the connect opcode. The signals of all the outlets connected to an inlet are combined in the inlet.

## See Also

*outleta outletk outletkid outletf inleta inletk inletkid connect alwayson ftgenonce*

## Credits

By: Michael Gogins 2009

# inletv

`inletv` — Receives an arate array signal into an instrument through a named port.

## Description

Receives an arate array signal into an instrument through a named port.

## Syntax

```
array inletv Sname
```

## Initialization

*Sname* -- String name of the inlet port. The name of the inlet is implicitly qualified by the instrument name or number, so it is valid to use the same inlet name in more than one instrument (but not to use the same inlet name twice in one instrument).

## Performance

*array* -- audio rate array inlet signal

During performance, the arate array inlet signal is received from each instance of an instrument containing an outlet port to which this inlet has been connected using the connect opcode. The signals of all the outlets connected to an inlet are summed in the inlet. The ports may have any number of channels, but the inlet port must have the same number of channels as the connected outlet ports.

## See Also

*outleta outletf outletk outletkid outletv inleta inletk inletkid inletf connect alwayson ftgenonce*

More information on this opcode: <http://www.csoundjournal.com/issue13/signalFlowGraphOpcodes.html>, written by Michael Gogins

## Credits

By: Michael Gogins 2009

# ino

ino — Reads eight-channel audio data from an external device or stream.

## Description

Reads eight-channel audio data from an external device or stream.



### Warning

This opcode is designed to be used only with orchestras that have `nchnls_i=8`. Doing so with orchestras with `nchnls_i > 8` will cause incorrect audio input.

## Syntax

```
ar1, ar2, ar3, ar4, ar5, ar6, ar7, ar8 ino
```

## Performance

Reads eight-channel audio data from an external device or stream. If the command-line `-i` flag is set, sound is read continuously from the audio input stream (e.g. `stdin` or a soundfile) into an internal buffer. Any number of these opcodes can read freely from this buffer.

## See Also

*diskin, in, inh, inq, ins, soundin*

## Credits

Author: John ffitch

# inq

inq — Reads quad audio data from an external device or stream.

## Description

Reads quad audio data from an external device or stream.



### Warning

This opcode is designed to be used only with orchestras that have `nchnls_i=4`. Doing so with orchestras with `nchnls_i > 4` will cause incorrect audio input.

## Syntax

```
ar1, ar2, ar3, a4 inq
```

## Performance

Reads quad audio data from an external device or stream. If the command-line `-i` flag is set, sound is read continuously from the audio input stream (e.g. `stdin` or a soundfile) into an internal buffer. Any number of these opcodes can read freely from this buffer.

## Examples

Here is an example of the `inq` opcode. It uses the file `inq.csd` [examples/inq.csd].

### Example 432. Example of the `inq` opcode.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -idac ;;realtime audio I/O
; For Non-realtime ouput leave only the line below:
; inq.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr      = 44100
ksmps   = 32
nchnls  = 2 ;2 channels out
0dbfs   = 1
nchnls_i = 4 ;4 channels in

instr 1 ;4 channels in, two channels out

ain1, ain2, ain3, ain4 inq ;grab your mics and sing

adel1 linseg 0, p3*.5, 0.02, p3*.5, 0 ;max delay time = 20ms
adel2 linseg 0.02, p3*.5, 0, p3*.5, 0.02 ;max delay time = 20ms
aout1 flanger ain1, adel1, .7
aoutr flanger ain2, adel2*.8
aoutla flanger ain3, adel2, .9
aoutra flanger ain4, adel2*.5

```

```
;write to quad file, 16 bits with header
fout "in_4.wav", 14, aoutl, aoutr, aoutla, aoutra
outs (aoutl+aoutla)*.5, (aoutr+aoutra)*.5 ;stereo out

endin
</CsInstruments>
<CsScore>

i 1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*diskin, in, inh, inl, ino, ins, soundin*

## Credits

Authors: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

# inrg

*inrg* — Allow input from a range of adjacent audio channels from the audio input device

## Description

*inrg* reads audio from a range of adjacent audio channels from the audio input device.

## Syntax

```
inrg kstart, ain1 [,ain2, ain3, ..., ainN]
```

## Performance

*kstart* - the number of the first channel of the input device to be accessed (channel numbers starts with 1, which is the first channel)

*ain1*, *ain2*, ... *ainN* - the output arguments filled with the incoming audio coming from corresponding channels.

*inrg* allows input from a range of adjacent channels from the input device. *kstart* indicates the first channel to be accessed (channel 1 is the first channel). The user must be sure that the number obtained by summing *kstart* plus the number of accessed channels -1 is  $\leq nchnls_i$ .



### Note

Note that this opcode is exceptional in that it produces its “output” on the parameters to the right.

## Credits

Author: Gabriel Maldonado

New in version 5.06

# ins

ins — Reads stereo audio data from an external device or stream.

## Description

Reads stereo audio data from an external device or stream.



### Warning

This opcode is designed to be used only with orchestras that have `nchnls_i=2`. Doing so with orchestras with `nchnls_i > 2` will cause incorrect audio input.

## Syntax

```
ar1, ar2 ins
```

## Performance

Reads stereo audio data from an external device or stream. If the command-line `-i` flag is set, sound is read continuously from the audio input stream (e.g. `stdin` or a soundfile) into an internal buffer. Any number of these opcodes can read freely from this buffer.

## Examples

Here is an example of the `ins` opcode. It uses the file `ins.csd` [examples/ins.csd].

### Example 433. Example of the `ins` opcode.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -idac  ;;realtime audio I/O
; For Non-realtime ouput leave only the line below:
; ins.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2 ;two channels out
0dbfs = 1

instr 1

ain1, ainr ins ;grab your mic and sing
adel linseg 0, p3*.5, 0.02, p3*.5, 0 ;max delay time = 20ms
aoutl flanger ain1, adel, .7
aoutr flanger ain1, adel*2, .8
fout "in_s.wav", 14, aoutl, aoutr ;write to stereo file,
outs aoutl, aoutr ;16 bits with header

endin
</CsInstruments>

```

```
<CsScore>  
  
i 1 0 10  
e  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*diskin, in, inh, inl, ino, inq, soundin mp3in,*

## Credits

Authors: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

# insremot

*insremot* — An opcode which can be used to implement a remote orchestra. This opcode will send note events from a source machine to one destination.

## Description

With the *insremot* and *insglobal* opcodes you are able to perform instruments on remote machines and control them from a master machine. The remote opcodes are implemented using the master/client model. All the machines involved contain the same orchestra but only the master machine contains the information of the score. During the performance the master machine sends the note events to the clients. The *insremot* opcode will send events from a source machine to one destination if you want to send events to many destinations (broadcast) use the *insglobal* opcode instead. These two opcodes can be used in combination.

## Syntax

```
insremot idestination, isource, instrnum [,instrnum...]
```

## Initialization

*idestination* -- a string that is the intended client computer (e.g. 192.168.0.100). This is the destination host which receives the events from the given instrument.

*isource* -- a string that is the intended server computer (e.g. 192.168.0.100). This is the source host which generates the events of the given instrument and sends it to the address given by *idestination*.

*instrnum* -- a list of instrument numbers which will be played on the destination machine

## Performance



### Note

If is essential the computers using this opcode have the same byte-order, data size (double or float) and pointer size. One cannot use it with mixed 32 and 64 computers for instance.



### Note

Internally this opcode may make use of the `gethostname` and `gethostbyname` functions to determine the client's and server's IP address for checking which messages are for which machine, or on non-windows system other techniques. If a computer has more than one IP address there is no way to control which IP address is found (but see below). On non-windows systems the default network is `eth0` or `en0`, and if that fails `wlan0`. (Since version 6.05) the network to be used can be set with an environment variable `CS_ETHER`.



### Note

The remote operation does not allow the sending of strings at all.

## Examples

Here is an example of the *insremot* opcode. It uses the files *insremot.csd* [examples/insremot.csd] and *insremotM.csd* [examples/insremotM.csd].

**Example 434. Example of the insremot opcode.**

The simple example below shows the bilbar example played on a remote machine. The master machine is named "192.168.1.100" and the client "192.168.1.101". Start the client on the machine (it will wait to receive the events from the master machine) and then start the master. Here is the command on linux to start a client (csound -+rtaudio=alsa -odac -dm0 insremot.csd), and the command on the master machine will look like this (csound -+rtaudio=alsa -odac -dm0 insremotM.csd).

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>

<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o insremot.wav -W ;; for file output any platform
</CsOptions>

<CsInstruments>
nchnls = 1

insremot "192.168.1.100", "192.168.1.101", 1

instr 1
  aq barmodel 1, 1, p4, 0.001, 0.23, 5, p5, p6, p7
    out      aq
endin

</CsInstruments>

<CsScore>
f0 360

e
</CsScore>

</CsoundSynthesizer>

<CsoundSynthesizer>

<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o insremotM.wav -W ;; for file output any platform
</CsOptions>

<CsInstruments>
nchnls = 1

insremot "192.168.1.100", "192.168.1.101", 1

instr 1
  aq barmodel 1, 1, p4, 0.001, 0.23, 5, p5, p6, p7
    out      aq
endin

</CsInstruments>

```

```
<CsScore>
i1 0 0.5 3 0.2 500 0.05
i1 0.5 0.5 -3 0.3 1000 0.05
i1 1.0 0.5 -3 0.4 1000 0.1
i1 1.5 4.0 -3 0.5 800 0.05
e
</CsScore>

</CsoundSynthesizer>
```

## See also

*inglobal, midglobal, midremot, remoteport*

## Credits

Author: Simon Schampijer  
2006

New in version 5.03

# insglobal

*insglobal* — An opcode which can be used to implement a remote orchestra. This opcode will send note events from a source machine to many destinations.

## Description

With the *insremot* and *insglobal* opcodes you are able to perform instruments on remote machines and control them from a master machine. The remote opcodes are implemented using the master/client model. All the machines involved contain the same orchestra but only the master machine contains the information of the score. During the performance the master machine sends the note events to the clients. The *insglobal* opcode sends the events to all the machines involved in the remote concert. These machines are determined by the *insremot* definitions made above the *insglobal* command. To send events to only one machine use *insremot*.

## Syntax

```
insglobal isource, instrnum [,instrnum...]
```

## Initialization

*isource* -- a string that is the intended server computer (e.g. 192.168.0.100). This is the source host which generates the events of the given instrument(s) and sends it to all the machines involved in the remote concert.

*instrnum* -- a list of instrument numbers which will be played on the destination machines

## Performance



### Note

If is essential the computers using this opcode have the same byte-order, data size (double or float) and pointer size. One cannot use it with mixed 32 and 64 computers for instance.



### Note

Internally this opcode makes use of the `gethostname` and `gethostbyname` functions to determine the client's and server's IP address for checking which messages are for which machine. If a computer has more than one IP address there is no way to control which IP address is found.



### Note

The remote operation does not allow the sending of strings at all.

## Examples

See the entry for *insremot* for an example of usage.

## See also

*insremot*, *midglobal*, *midremot*, *remoteport*

## Credits

Author: Simon Schampijer  
2006

New in version 5.03

# instr

instr — Starts an instrument block.

## Description

Starts an instrument block.

## Syntax

```
instr i, j, ...
```

## Initialization

Starts an instrument block defining instruments *i, j, ...*

*i, j, ...* must be numbers, not expressions. Any positive integer is legal, and in any order, but excessively high numbers are best avoided.



### Note

There may be any number of instrument blocks in an orchestra.

Instruments can be defined in any order (but they will always be both initialized and performed in ascending instrument number order, with the exception of notes triggered by real time events that are initialized in the order of being received but still performed in ascending instrument number order). Instrument blocks cannot be nested (i.e. one block cannot contain another).

## Examples

Here is an example of the instr opcode. It uses the file *instr.csd* [examples/instr.csd].

### Example 435. Example of the instr opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc    ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o instr.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1
```

```
; Instrument #1.
instr 1
  iamp = 10000
  icps = 440
  iphs = 0

  a1 oscils iamp, icps, iphs
  out a1
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for 2 seconds.
i 1 0 2
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*endin, in, out, opcode, endop, setksmps, xin, xout, subinstr, subinstrinit*

## Credits

Example written by Kevin Conder.

# int

int — Extracts an integer from a decimal number.

## Description

Returns the integer part of  $x$ .

## Syntax

```
int(x) (init-rate or control-rate; also works at audio rate in Csound5)
```

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

## Examples

Here is an example of the int opcode. It uses the file *int.csd* [examples/int.csd].

### Example 436. Example of the int opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o int.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

icount init 0
loop:
  inum = icount / 3
  inm = int(inum)
  prints "integer (%f/3) = %f\\n", icount, inm
loop_lt icount, 1, 10, loop

endin
</CsInstruments>
<CsScore>

i 1 0 0
e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like these:

```
integer (0.000000/3) = 0.000000  
integer (1.000000/3) = 0.000000  
integer (2.000000/3) = 0.000000  
integer (3.000000/3) = 1.000000  
integer (4.000000/3) = 1.000000  
integer (5.000000/3) = 1.000000  
integer (6.000000/3) = 2.000000  
integer (7.000000/3) = 2.000000  
integer (8.000000/3) = 2.000000  
integer (9.000000/3) = 3.000000
```

## See Also

*abs, exp, frac, log, log10, i, sqrt*

## Credits

# integ

integ — Modify a signal by integration.

## Description

Modify a signal by integration.

## Syntax

```
ares integ asig [, iskip]
kres integ ksig [, iskip]
```

## Initialization

*iskip* (optional) -- initial disposition of internal save space (see *reson*). The default value is 0.

## Performance

*integ* and *diff* perform integration and differentiation on an input control signal or audio signal. Each is the converse of the other, and applying both will reconstruct the original signal. Since these units are special cases of low-pass and high-pass filters, they produce a scaled (and phase shifted) output that is frequency-dependent. Thus *diff* of a sine produces a cosine, with amplitude  $2 * \pi * \text{Hz} / \text{sr}$  that of the original (for each component partial); *integ* will inversely affect the magnitudes of its component inputs. With this understanding, these units can provide useful signal modification.

## Examples

Here is an example of the integ opcode. It uses the file *integ.csd* [examples/integ.csd].

### Example 437. Example of the integ opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o integ.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

asig diskin2 "fox.wav", 1
```

```
        outs asig, asig

    endin

    instr 2 ; with diff

    asig diskin2 "fox.wav", 1
    ares diff asig
        outs ares, ares

    endin

    instr 3 ; with integ

    asig diskin2 "fox.wav", 1
    aint integ asig
    aint = aint*.05 ;way too loud
        outs aint, aint

    endin

    instr 4 ; with diff and integ

    asig diskin2 "fox.wav", 1
    ares diff asig
    aint integ ares
        outs aint, aint

    endin

</CsInstruments>
<CsScore>

i 1 0 1
i 2 1 1
i 3 2 1
i 4 3 1

e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*diff, downsamp, interp, samphold, upsamp*

# interp

interp — Converts a control signal to an audio signal using linear interpolation.

## Description

Converts a control signal to an audio signal using linear interpolation.

## Syntax

```
ares interp ksig [, iskip] [, imode]  
      [, ivalue]
```

## Initialization

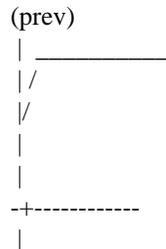
*iskip* (optional, default=0) -- if non zero skips initialisation of internal save space (see *reson*).

*imode* (optional, default=0) -- sets the initial output value to the first k-rate input instead of zero. The following graphs show the output of *interp* with a constant input value, in the original, when skipping init, and in the new mode:

### Example 438. iskip=0, imode=0

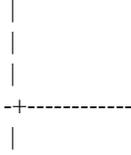


### Example 439. iskip=1, imode=0



### Example 440. iskip=0, imode=1





*ivalue* (optional, default=0) -- initial value if both imode and iskip are zero.

## Performance

*ksig* -- input k-rate signal.

*interp* converts a control signal to an audio signal. It uses linear interpolation between successive kvals.

## Examples

Here is an example of the *interp* opcode. It uses the file *interp.csd* [examples/interp.csd].

### Example 441. Example of the *interp* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o interp.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 8000
kr = 8
ksmps = 1000
nchnls = 1

; Instrument #1 - a simple instrument.
instr 1
; Create an amplitude envelope.
kamp linseg 0, p3/2, 20000, p3/2, 0

; The amplitude envelope will sound rough because it
; jumps every ksmps period, 1000.
a1 oscil kamp, 440, 1
out a1
endin

; Instrument #2 - a smoother sounding instrument.
instr 2
; Create an amplitude envelope.
kamp linseg 0, p3/2, 25000, p3/2, 0
aamp interp kamp

; The amplitude envelope will sound smoother due to
; linear interpolation at the higher a-rate, 8000.
a1 oscil aamp, 440, 1
```

```
    out a1
  endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 256 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
; Play Instrument #2 for two seconds.
i 2 2 2
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*diff*, *downsamp*, *integ*, *samphold*, *upsamp*

More information on this opcode: <http://www.csoundjournal.com/issue10/CsoundRates.html> , written by Andrés Cabrera

## Credits

Updated November 2002, thanks to a note from both Rasmus Ekman and Istvan Varga.

# invalue

invalue — Reads a k-rate signal from a user-defined channel.

## Description

Reads a k-rate or i-rate signal or string from a user-defined channel.

## Syntax

```
invalue invalue "channel name"  
kvalue invalue "channel name"  
Sname invalue "channel name"
```

## Performance

*invalue*, *kvalue* -- The value that is read from the channel.

*Sname* -- The string variable that is read from the channel.

"*channel name*" -- An integer, string (in double-quotes), or string variable identifying the channel.

## Examples

Here is an example of the invalue opcode. It uses the file *invalue.csd* [examples/invalue.csd].

### Example 442. Example of the invalue opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
</CsOptions>  
<CsInstruments>  
;run this example in CsoundQt, a Csound editor that provides widgets  
;make the Widgets-panel visible, by clicking the Widgets symbol in the menu or pressing (Alt+1).  
  
sr = 44100  
ksmps = 32  
nchnls = 2  
0dbfs = 1  
; written by Andres Cabrera  
instr 1  
  
kfreq invalue "freq" ; Quotes are needed here  
asig oscil 0.1, kfreq, 1  
     outs asig, asig  
  
endin  
</CsInstruments>  
<CsScore>  
f 1 0 1024 10 1 ;sine  
i 1 0 300 ;play for 300 seconds
```

```
e  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*outvalue*

## Credits

Author: Matt Ingalls

i-rate version new in Csound 6.04

# inx

*inx* — Reads a 16-channel audio signal from an external device or stream.

## Description

Reads a 16-channel audio signal from an external device or stream.



### Warning

This opcode is designed to be used only with orchestras that have `nchnls=16`. Doing so with orchestras with `nchnls > 16` will cause incorrect audio input.

## Syntax

```
ar1, ar2, ar3, ar4, ar5, ar6, ar7, ar8, ar9, ar10, ar11, ar12, \  
ar13, ar14, ar15, ar16 inx
```

## Performance

*inx* reads a 16-channel audio signal from an external device or stream. If the command-line `-i` flag is set, sound is read continuously from the audio input stream (e.g. *stdin* or a soundfile) into an internal buffer.

## See Also

*in32*, *inch*, *inz*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2000

New in Csound Version 4.07

## inz

inz — Reads multi-channel audio samples into a ZAK array from an external device or stream.

### Description

Reads multi-channel audio samples into a ZAK array from an external device or stream.

### Syntax

```
inz ksig1
```

### Performance

*inz* reads audio samples in *nchnls* into a ZAK array starting at *ksig1*. If the command-line *-i* flag is set, sound is read continuously from the audio input stream (e.g. *stdin* or a soundfile) into an internal buffer.

### See Also

*in32*, *inch*, *inx*

### Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2000

New in Csound Version 4.07

# JackoAudioIn

JackoAudioIn — Receives an audio signal from a Jack port.



## Note

plugin: Requires jacko plugin

## Description

Receives an audio signal from a Jack audio input port inside this instance of Csound, which in turn has received the signal from its connected external Jack audio output port.

## Syntax

```
asignal JackoAudioIn ScsoundPortName
```

## Initialization

*ScsoundPortName* -- The short name ("portname") of the internal Jack audio input port.

## Performance

*asignal* -- Audio received from the external Jack output port to which *ScsoundPortName* is connected.

## See Also

*JackoInfo*, *JackoInfo*, *JackoFreewheel*, *JackoAudioOutConnect*, *JackoAudioOutConnect*, *JackoMidiInConnect*, *JackoMidiOutConnect*, *JackoOn*, *JackoAudioOut*, *JackoMidiOut*, *JackoNoteOut*, *JackoTransport*.

## Credits

By: Michael Gogins 2010

# JackoAudioInConnect

JackoAudioInConnect — Creates an audio connection from a Jack port to Csound.



## Note

plugin: Requires jacko plugin

## Description

In the orchestra header, creates an audio connection from an external Jack audio output port to a Jack audio input port inside this instance of Csound.

## Syntax

```
JackoAudioInConnect SexternalPortName, ScsoundPortName
```

## Initialization

*SexternalPortName* -- The full name ("clientname:portname") of an external Jack audio output port.

*ScsoundPortName* -- The short name ("portname") of the internal Jack audio input port.

## Performance

The actual audio must be read with the JackoAudioIn opcode.

## See Also

*JackoInfo*, *JackoInfo*, *JackoFreewheel*, *JackoAudioOutConnect*, *JackoMidiInConnect*, *JackoMidiOutConnect*, *JackoOn*, *JackoAudioIn*, *JackoAudioOut*, *JackoMidiOut*, *JackoNoteOut*, *JackoTransport*.

## Credits

By: Michael Gogins 2010

# JackoAudioOut

JackoAudioOut — Sends an audio signal to a Jack port.



## Note

plugin: Requires jacko plugin

## Description

Sends an audio signal to an internal Jack audio output port, and in turn to its connected external Jack audio input port.

Note that it is possible to send audio out via Jack to the system audio interface, while at the same time rendering to a regular Csound output soundfile.

## Syntax

```
JackoAudioOut ScsoundPortName, asignal
```

## Initialization

*ScsoundPortName* -- The short name ("portname") of the internal Jack audio output port.

## Performance

*asignal* -- Audio to be sent to the external Jack audio input port to which CsoundPortName is connected.

Audio from multiple instances of the opcode sending to the same Jack port is summed before sending.

## See Also

*JackoInfo*, *JackoInfo*, *JackoFreewheel*, *JackoAudioOutConnect*, *JackoMidiInConnect*, *JackoMidiOutConnect*, *JackoOn*, *JackoAudioIn*, *JackoMidiOut*, *JackoMidiOut*, *JackoTransport*.

## Credits

By: Michael Gogins 2010

# JackoAudioOutConnect

JackoAudioOutConnect — Creates an audio connection from Csound to a Jack port.



## Note

plugin: Requires jacko plugin

## Description

In the orchestra header, creates an audio connection from a Jack audio output port inside this instance of Csound to an external Jack audio input port.

## Syntax

```
JackoAudioOutConnect ScsoundPortName, SexternalPortName
```

## Initialization

*ScsoundPortName* -- The short name ("portname") of the internal Jack audio output port.

*SexternalPortName* -- The full name ("clientname:portname") of an external Jack audio input port.

## Performance

The actual audio must be written with the JackoAudioOut opcode.

## See Also

*The Jacko Opcodes, JackoInfo, JackoInfo, JackoFreewheel, JackoAudioOutConnect, JackoMidiInConnect, JackoMidiOutConnect, JackoOn, JackoAudioIn, JackoAudioOut, JackoMidiOut, JackoNoteOut, JackoTransport.*

## Credits

By: Michael Gogins 2010

# JackoFreewheel

JackoFreewheel — Turns Jack's freewheeling mode on or off.



## Note

plugin: Requires jacko plugin

## Description

Turns Jack's freewheeling mode on or off.

When freewheeling is on, if supported by the rest of the Jack system, Csound will run as fast as possible, which may be either faster or slower than real time.

This is essential for rendering scores that are too dense for real-time performance to a soundfile, without xruns or dropouts.

## Syntax

```
JackoFreewheel [ienabled]
```

## Initialization

*ienabled* -- Turns freewheeling on (the default) or off.

## See Also

*JackoInit*, *JackoInfo*, *JackoAudioInConnect*, *JackoAudioOutConnect*, *JackoMidiInConnect*, *JackoMidiOutConnect*, *JackoOn*, *JackoAudioIn*, *JackoAudioOut*, *JackoMidiOut*, *JackoNoteOut*, *JackoTransport*.

## Credits

By: Michael Gogins 2010

# JackoInfo

JackoInfo — Prints information about the Jack system.



## Note

plugin: Requires jacko plugin

## Description

Prints the Jack daemon and client names, the sampling rate and frames per period, and all active Jack port names, types, states, and connections.

## Syntax

```
JackoInfo
```

## Initialization

May be called any number of times in the orchestra header, for example both before and after creating Jack ports in the Csound orchestra header.

## Examples

Here is an example of the JackoInfo opcode. It uses the file *JackInfo.csd* [examples/JackoInfo.csd].

### Example 443. Example of the JackoInfo opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-n
</CsOptions>
<CsInstruments>

sr = 48000
ksmps = 128
nchnls = 2
0dbfs = 1

instr 1

JackoInit "default", "csound"
JackoInfo

endin
</CsInstruments>
<CsScore>

i 1 0 0
e
</CsScore>
```

`</CsoundSynthesizer>`

## See Also

*JackoInit, JackoFreewheel, JackoAudioInConnect, JackoAudioOutConnect, JackoMidiInConnect, JackoMidiOutConnect, JackoOn, JackoAudioIn, JackoAudioOut, JackoMidiOut, JackoNoteOut, JackoTransport.*

## Credits

By: Michael Gogins 2010

# JackoInit

JackoInit — Initializes Csound as a Jack client.



## Note

plugin: Requires jacko plugin

## Description

Initializes this instance of Csound as a Jack client.

Csound's `sr` must be equal to the Jack daemon's frames per second.

Csound's `ksmps` must be equal to the Jack daemon's frames per period.

Frames per period must not only (a) be a power of 2, but also (b) go evenly into the frames per second, e.g. 128 frames per period goes into 48000 frames per second 375 times, for a latency or MIDI time granularity of about 2.7 milliseconds (as good as or better than the absolute best human performers).

The order of processing of all signals that pass from Jack input ports, through Csound processing, and to Jack output ports, must be properly determined by sequence of instrument and opcode definition within Csound.

## Syntax

```
JackoInit ServerName, SclientName
```

## Initialization

*SclientName* -- The name of the Jack client; normally, should be "csound".

*ServerName* -- The name of the Jack daemon; normally, will be "default".

This opcode must be called once and only once in the orchestra header, and before any other Jack opcodes. If more than one instance of Csound is using the Jack opcodes at the same time, then each instance of Csound must use a different client name.

## Examples

Here is an example of the JackoInit opcode. It uses the file *JackoInit.csd* [examples/JackoInit.csd].

### Example 444. Example of the JackoInit opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
-n  
</CsOptions>  
<CsInstruments>
```

```
sr = 48000
ksmps = 128
nchnls = 2
0dbfs = 1

instr 1

JackoInit "default", "csound"
JackoInfo

endin
</CsInstruments>
<CsScore>

i 1 0 0
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*JackoInfo, JackoFreewheel, JackoAudioInConnect, JackoAudioOutConnect, JackoMidiInConnect, JackoMidiOutConnect, JackoOn, JackoAudioIn, JackoAudioOut, JackoMidiOut, JackoNoteOut, JackoTransport.*

## Credits

By: Michael Gogins 2010

# JackoMidiInConnect

JackoMidiInConnect — Creates a MIDI connection from a Jack port to Csound.



## Note

plugin: Requires jacko plugin

## Description

In the orchestra header, creates a MIDI connection from an external Jack MIDI output port to this instance of Csound.

## Syntax

```
JackoMidiInConnect SexternalPortName, ScsoundPortName
```

## Initialization

*SexternalPortName* -- The full name ("clientname:portname") of an external Jack MIDI output port.

*ScsoundPortName* -- The short name ("portname") of the internal Jack MIDI input port.

Must be used in conjunction with the `-M0 --rtmidi=null` Csound command-line options. Can be used in with the MIDI inter-operability command-line options and/or opcodes to enable the use of ordinary Csound instrument definitions to render external scores or MIDI sequences.

Note that Csound can connect to ALSA ports through Jack, but in that case you will have to identify the port by its alias in the JackInfo printout.

## Performance

The actual MIDI events will be received in the regular Csound way, i.e. through a MIDI driver and the sensevents mechanism, rather than through a Jack input port opcode.

The granularity of timing is Csound's kperiod.

## See Also

*JackoInfo*, *JackoInfo*, *JackoFreewheel*, *JackoAudioOutConnect*, *JackoAudioOutConnect*, *JackoMidiOutConnect*, *JackoOn*, *JackoAudioIn*, *JackoAudioOut*, *JackoMidiOut*, *JackoNoteOut*, *JackoTransport*.

## Credits

By: Michael Gogins 2010

# JackoMidiOutConnect

JackoMidiOutConnect — Creates a MIDI connection from Csound to a Jack port.



## Note

plugin: Requires jacko plugin

## Description

In the orchestra header, creates a connection from a Jack MIDI output port inside this instance of Csound to an external Jack MIDI input port.

## Syntax

```
JackoMidiOutConnect ScsoundPortName, SexternalPortName
```

## Initialization

*ScsoundPortName* -- The short name ("portname") of the internal Jack MIDI output port.

*SexternalPortName* -- The full name ("clientname:portname") of an external Jack MIDI input port.

## Performance

The actual MIDI data must be written with the JackoMidiOut or JackoNoteOut opcodes.

## See Also

*JackoInfo JackoInfo JackoFreewheel JackoAudioOutConnect JackoMidiInConnect JackoMidiOutConnect JackoOn JackoAudioIn JackoAudioOut JackoMidiOut JackoNoteOut JackoTransport*

## Credits

By: Michael Gogins 2010

# JackoMidiOut

JackoMidiOut — Sends a MIDI channel message to a Jack port.



## Note

plugin: Requires jacko plugin

## Description

Sends a MIDI channel message to a Jack MIDI output port inside this instance of Csound, and in turn to its connected external Jack MIDI input port.

## Syntax

```
JackoMidiOut ScsoundPortName, kstatus, kchannel, kdata1[, kdata2]
```

## Initialization

*ScsoundPortName* -- The short name ("portname") of the internal Jack MIDI output port.

## Performance

*kstatus* -- MIDI status byte; must indicate a MIDI channel message.

*kchannel* -- MIDI channel (from 0 through 15).

*kdata1* -- First data byte of a MIDI channel message.

*kdata2* -- Optional second data byte of a MIDI channel message.

This opcode can be called any number of times in the same kperiod. Messages from multiple instances of the opcode sending to the same port are collected before sending.

Running status, system exclusive messages, and real-time messages are not supported.

The granularity of timing is Csound's kperiod.

## See Also

*JackoInfo*, *JackoInfo*, *JackoFreewheel*, *JackoAudioOutConnect*, *JackoMidiInConnect*, *JackoMidiOutConnect*, *JackoOn*, *JackoAudioIn*, *JackoNoteOut*, *JackoTransport*.

## Credits

By: Michael Gogins 2010

# JackoNoteOut

JackoNoteOut — Sends a MIDI channel message to a Jack port.



## Note

plugin: Requires jacko plugin

## Description

Sends a MIDI channel message to a Jack MIDI output port inside this instance of Csound, and in turn to its connected external Jack MIDI input port.

## Syntax

```
JackoNoteOut ScsoundPortName, kstatus, kchannel, kdata1[, kdata2]
```

## Initialization

*ScsoundPortName* -- The short name ("portname") of the internal Jack MIDI output port.

## Performance

*kstatus* -- MIDI status byte; must indicate a MIDI channel message.

*kchannel* -- MIDI channel (from 0 through 15).

*kdata1* -- First data byte of a MIDI channel message.

*kdata2* -- Optional second data byte of a MIDI channel message.

This opcode can be called any number of times in the same kperiod. Messages from multiple instances of the opcode sending to the same port are collected before sending.

Running status, system exclusive messages, and real-time messages are not supported.

The granularity of timing is Csound's kperiod.

## See Also

*JackoInfo*, *JackoInfo*, *JackoFreewheel*, *JackoAudioInConnect*, *JackoAudioOutConnect*, *JackoMidiInConnect*, *JackoMidiOutConnect*, *JackoOn*, *JackoAudioIn*, *JackoMidiOut*, *The Jacko Opcodes*.

## Credits

By: Michael Gogins 2010

# JackoOn

JackoOn — Enables or disables all Jack ports.



## Note

plugin: Requires jacko plugin

## Description

In the orchestra header, after all Jack connections have been created, enables or disables all Jack input and output opcodes inside this instance of Csound to read or write data.

## Syntax

```
JackoOn [iactive]
```

## Initialization

*iactive* -- A flag that turns the ports on (the default) or off.

## See Also

*JackoInit, JackoFreewheel, JackoAudioInConnect, JackoAudioOutConnect, JackoMidiInConnect, JackoMidiOutConnect, JackoAudioIn, JackoAudioOut, JackoMidiOut, JackoNoteOut, JackoTransport.*

## Credits

By: Michael Gogins 2010

# JackoTransport

JackoTransport — Control the Jack transport.



## Note

plugin: Requires jacko plugin

## Description

Starts, stops, or repositions the Jack transport. This is useful, e.g., for starting an external sequencer playing to send MIDI messages to Csound.

## Syntax

```
JackoTransport kcommand, [kposition]
```

## Performance

*kcommand* -- 0 means "no action", 1 starts the transport, 2 stops the transport, and 3 positions the transport to *kposition* seconds from the beginning of performance (i.e. time 0 in the score).

*kposition* -- Time to position to the transport, in seconds from the beginning of performance (i.e. time 0 in the score).

This opcode can be used at init time or during performance.

The granularity of timing is Csound's *kperiod*.

## See Also

*JackoInfo*, *JackoInfo*, *JackoFreewheel*, *JackoAudioOutConnect*, *JackoMidiInConnect*, *JackoMidiOutConnect*, *JackoOn*, *JackoAudioIn*, *JackoMidiOut*, *JackoNoteOut*.

## Credits

By: Michael Gogins 2010

# jacktransport

jacktransport — Start/stop jack\_transport and can optionally relocate the playback head.



## Note

plugin: Requires jackTransport plugin

## Description

Start/stop jack\_transport and can optionally relocate the playback head.

## Syntax

```
jacktransport icommand [, ilocation]
```

## Initialization

*icommand* -- 1 to start playing, 0 to stop.

*ilocation* -- optional location in seconds to specify where the playback head should be moved. If omitted, the transport is started from current location.



## Note

Since *jacktransport* depends on jack audio connection kit, it will work only on Linux or Mac OS X systems which have the jack server running.

## Examples

Here is a simple example of the jacktransport opcode. It uses the file *jacktransport.csd* [examples/jack-transport.csd].

### Example 445. Simple example of the jacktransport opcode.

```
<CsoundSynthesizer>

<CsOptions>
+rtaudio=JACK -b 64 --sched -o dac:system:playback_
</CsOptions>

<CsInstruments>
sr      =      44100
ksmps  =      16
nchnls = 2

instr 1
jacktransport p4, p5
endin

instr 2
jacktransport p4
endin
```

```
</CsInstruments>
<CsScore>

i2 0 5 1; play
i2 5 1 0; stop
i1 6 5 1 2 ; move at 2 seconds and start playing back
i1 11 1 0 0 ; stop and rewind

e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Cesare Marilungo

New in version 5.08

# jitter

`jitter` — Generates a segmented line whose segments are randomly generated.

## Description

Generates a segmented line whose segments are randomly generated.

## Syntax

```
kout jitter kamp, kcpsMin, kcpsMax
```

## Performance

*kamp* -- Amplitude of jitter deviation

*kcpsMin* -- Minimum speed of random frequency variations (expressed in cps)

*kcpsMax* -- Maximum speed of random frequency variations (expressed in cps)

*jitter* generates a segmented line whose segments are randomly generated inside the `+kamp` and `-kamp` interval. Duration of each segment is a random value generated according to `kcpsmin` and `kcpsmax` values.

*jitter* can be used to make more natural and “analog-sounding” some static, dull sound. For best results, it is suggested to keep its amplitude moderate.

## Examples

Here is an example of the `jitter` opcode. It uses the file `jitter.csd` [examples/jitter.csd].

### Example 446. Example of the `jitter` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o jitter.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kamp      init p4
kcpsmin   init 4
kcpsmax   init 8
```

```
kj2 jitter kamp, kcpsmin, kcpsmax
aout pluck 1, 200+kj2, 1000, 0, 1
aout dcblock aout ;remove DC
outs aout, aout

endin
</CsInstruments>
<CsScore>

i 1 0 15 2 ;a bit jitter
i 1 8 15 10 ;some more
i 1 16 15 20 ;lots more
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*jitter2, vibr, vibrato*

## Credits

Author: Gabriel Maldonado

New in Version 4.15

# jitter2

`jitter2` — Generates a segmented line with user-controllable random segments.

## Description

Generates a segmented line with user-controllable random segments.

## Syntax

```
kout jitter2 ktotamp, kamp1, kcps1,  
      kamp2, kcps2, kamp3, kcps3[ , iopt]
```

## Initialisation

`iopt` -- Optional, controls time of onset of effects. If zero (default) the output is zero until the end of the shortest `kcps` value. If not zero the effect starts immediately

## Performance

`ktotamp` -- Resulting amplitude of `jitter2`

`kamp1` -- Amplitude of the first jitter component

`kcps1` -- Speed of random variation of the first jitter component (expressed in cps)

`kamp2` -- Amplitude of the second jitter component

`kcps2` -- Speed of random variation of the second jitter component (expressed in cps)

`kamp3` -- Amplitude of the third jitter component

`kcps3` -- Speed of random variation of the third jitter component (expressed in cps)

`jitter2` also generates a segmented line such as `jitter`, but in this case the result is similar to the sum of three `randi` opcodes, each one with a different amplitude and frequency value (see `randi` for more details), that can be varied at k-rate. Different effects can be obtained by varying the input arguments.

`jitter2` can be used to make more natural and “analog-sounding” some static, dull sound. For best results, it is suggested to keep its amplitude moderate.

## Examples

Here is an example of the `jitter2` opcode. It uses the file `jitter2.csd` [examples/jitter2.csd].

### Example 447. Example of the `jitter2` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
```

```
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o jitter2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ktotamp init p4
kamp1  init .5
kcps1  init 10
kamp2  init .5
kcps2  init 2
kamp3  init .5
kcps3  init 3

kj2 jitter2 ktotamp, kamp1, kcps1, kamp2, kcps2, kamp3, kcps3
aout pluck 1, 200+kj2, 1000, 0, 1
aout dcblock aout
      outs aout, aout

endin
</CsInstruments>
<CsScore>

i 1 0 15 2 ;a bit jitter
i 1 8 15 10 ;some more
i 1 16 15 20 ;lots more
e
</CsScore>
</CsSoundSynthesizer>
```

## See Also

*jitter, vibr, vibrato*

## Credits

Author: Gabriel Maldonado, John ffitc

New in Version 4.15

iopt new in version 6.08

# joystick

joystick — Reads data from a joystick controller.

## Description

Reads data from a Linux joystick controller

## Syntax

```
kres joystick kdevice ktab
```

## Performance



### Note

Please note that this opcode is currently only supported on Linux.

*kdevice* -- the index of the joystick device, either `/dev/jsN` or `/dev/input/jsN`.

*ktab* -- A table to hold input results, should be at least enough elements to store one value for each stick axis and one for each button + 2. The first two elements of the table are initialized with the number of axes and the number of buttons, respectively, when a joystick is opened. If a joystick is unplugged during performance, the opcode will repeatedly attempt to reopen the device with a delay between attempts.

## Examples

Here is an example of the joystick opcode. It uses the file *joystick.csd* [examples/joystick.csd].

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o joystick.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
;0dbfs = 1

instr 1 ; gives information about your joystick in real time

kmask    joystick    0, 1
kidx     = 2
kaxes    tab 0, 1 ; number of axes has been stored in position 0
kbuttons tab 1, 1 ; number of buttons has been stored in position 1
printf "this joystick has %d axes, %d buttons\n", kidx, kaxes, kbuttons
kuniq    init 0
```

```

reportaxis: ; first we see if we have any x/y input
kcheck = kmask & (1<<kidx)
if kcheck == 0 kgoto nexta
kres    tab    kidx, 1
kuniq   = kuniq + 1 ; to be sure to make the printf print
printf  "axis %d, value %6d\n", kuniq, kidx-2, kres
nexta:
kidx    = kidx+1
if kidx < (kaxes+2) kgoto reportaxis

reportbutton: ; now we check for any buttons pressed
kcheck = kmask & 1<<kidx
if kcheck == 0 kgoto nextb
kres    tab    kidx, 1 ; a button has been pressed, get from table
printf  "button %d, pushed\n", kidx*kres, (kidx-(kaxes+2))
printf  "button %d, released\n", kidx*(1-kres), (kidx-(kaxes+2))

nextb:
kidx    = kidx+1
if kidx < (kaxes+kbuttons+2) kgoto reportbutton

endin
</CsInstruments>
<CsScore>

f1 0 32 7 0 7 0 ; will hold the joystick data

i1 0 60000

e
</CsScore>
</CsoundSynthesizer>

```

Here is another example of the joystick opcode. It uses the file *joystick-2.csd* [examples/joystick-2.csd].

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o joystick-2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
;0dbfs = 1

instr 1

kmask joystick 0, 1
kaxes  init 0
kbuttons init 0
kx0    init 0 ; first two entries are # of axes and # of buttons,
ky0    init 0 ; then axes, then buttons
vtabk 0, 1, kaxes, kbuttons, kx0, ky0
kidx   = 2*kaxes

buttons:
kcheck = kmask & 1<<kidx ; if the button was just now pressed and...
kres   tab    kidx, 1 ; if button value is one, start a note
schedkwhen kres*kcheck, 1, 20, 2, 0, 60000, kidx, kx0, ky0
kidx   = kidx+1

```

```
if kidx < (kaxes+kbuttons+2) kgoto buttons

endin

instr 2 ; play a tone until the button is released

kstop  tab p4, 1 ; when this button is released, we fade out
ihz     init cpsoct(((p5+32767)/9362)+5) ; ~ 30 hz to 4khz
print  ihz
ito     init ampdb(((p6+32767)/2184)+60) ; ~ 60 - 90 db
kenv    init 0
kdelta  init ito/(kr*10)
if kstop == 1 kgoto output
if kdelta < 0 kgoto output
kdelta  = kdelta*-1

output:
  kenv = kenv+kdelta
  kenv limit kenv, 0, ito
  aout oscils 1, ihz, 0
  aout = kenv*aout
  outs aout, aout
if kenv != 0 kgoto noexit
if kdelta > 0 kgoto noexit
turnoff
noexit:

endin
</CsInstruments>
<CsScore>
f1 0 32 7 0 7 0 ; will hold the joystick data

i1 0 60000

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Justin Glenn Smith  
2010

New in version 5.17.12

# jspline

jspline — A jitter-spline generator.

## Description

A jitter-spline generator.

## Syntax

```
ares jspline xamp, kcpsMin, kcpsMax
```

```
kres jspline kamp, kcpsMin, kcpsMax
```

## Performance

*kres, ares* -- Output signal

*xamp* -- Amplitude factor

*kcpsMin, kcpsMax* -- Range of point-generation rate. Min and max limits are expressed in cps.

*jspline* (jitter-spline generator) generates a smooth curve based on random points generated at [*cpsMin*, *cpsMax*] rate. This opcode is similar to *randomi* or *randi* or *jitter*, but segments are not straight lines, but cubic spline curves. Output value range is approximately  $> -xamp$  and  $< xamp$ . Actually, real range could be a bit greater, because of interpolating curves between each pair of random-points.

At present time generated curves are quite smooth when *cpsMin* is not too different from *cpsMax*. When *cpsMin-cpsMax* interval is big, some little discontinuity could occur, but it should not be a problem, in most cases. Maybe the algorithm will be improved in next versions.

These opcodes are often better than *jitter* when user wants to “naturalize” or “analogize” digital sounds. They could be used also in algorithmic composition, to generate smooth random melodic lines when used together with *samphold* opcode.

Note that the result is quite different from the one obtained by filtering white noise, and they allow the user to obtain a much more precise control.

## Examples

Here is an example of the *jspline* opcode. It uses the file *jspline.csd* [examples/jspline.csd].

### Example 448. Example of the *jspline* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsoundOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
```

```
; -o jspline.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kamp    init p4
kcpsmin init 2
kcpsmax init 20

ksp    jspline kamp, kcpsmin, kcpsmax
aout   pluck 1, 200+ksp, 1000, 0, 1
aout   dcblock aout ;remove DC
       outs aout, aout

endin
</CsInstruments>
<CsScore>

i 1 0 10 2 ;a bit jitter
i 1 8 10 10 ;some more
i 1 16 10 20 ;lots more
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Gabriel Maldonado

New in Version 4.15

## k

k — Converts a i-rate parameter to an k-rate value. Or converts an a-rate value to a k-rate value by down-sampling.

### Description

Converts an i-rate value to control rate, for example to be used with *rnd()* and *birnd()* to generate random numbers at k-rate.

### Syntax

`k(x)` (i-rate args only)

`k(x)` (a-rate args only)

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

### See Also

*i a*

More information on this opcode: <http://www.csoundjournal.com/issue10/CsoundRates.html> , written by Andrés Cabrera

### Credits

Author: Istvan Varga

New in version Csound 5.00

Author: John ffitich

arate to krate new in version Csound 6.00

# K35\_hpf

K35\_hpf — Zero-delay feedback implementation of Korg35 resonant high-pass filter.

## Description

Zero-delay feedback implementation of Korg35 resonant high-pass filter. This filter design is found in the Korg MS10 early MS20.

## Syntax

```
asig K35_hpf ain, xcf, xQ [, inlp, isaturation, istor]
```

## Initialization

*istor* --initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*asig* -- output signal.

*ain* -- input signal.

*xcf* -- filter cutoff frequency (i-, k-, or a-rate).

*xQ* -- filter Q value (i-, k-, or a-rate). Range 1.0-10.0 (clamped by opcode). Self-oscillation occurs at 10.0.

*inlp* (optional, default=0) -- Non-linear processing method. 0 = no processing, 1 = non-linear processing. Method 1 uses  $\tanh(ksaturation * input)$ . Enabling NLP may increase the overall output of filter above unity and should be compensated for outside of the filter.

*ksaturation* (optional, default=1) -- saturation amount to use for non-linear processing. Values > 1 increase the steepness of the NLP curve.

## Examples

Here is an example of the K35\_hpf opcode. It uses the file *k35.csd* [examples/k35.csd].

### Example 449. Example of the K35\_hpf opcode.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
; =====
<CsInstruments>

sr = 48000
ksmps = 1
nchnls = 2
odbfs = 1

;; test instruments to demo filter cutoff sweep with high resonance
```

```
instr 1

asig = vco2(0.5, cps2pch(6.00, 12))
asig = K35_lpf(asig, expseg:a(10000, p3, 30), 9.9, 0, 1)
asig *= 0.25
asig = limit(asig, -1.0, 1.0)

outc(asig, asig)

endin

instr 2

asig = vco2(0.5, cps2pch(6.00, 12))
asig = K35_lpf(asig, expseg:k(10000, p3, 30), 9.9, 0, 1)
asig *= 0.25
asig = limit(asig, -1.0, 1.0)

outc(asig, asig)

endin

instr 3

asig = vco2(0.5, cps2pch(6.00, 12))
asig = K35_hpf(asig, expseg:a(10000, p3, 30), 9.9, 0, 1)
asig *= 0.25
asig = limit(asig, -1.0, 1.0)

outc(asig, asig)

endin

instr 4

asig = vco2(0.5, cps2pch(6.00, 12))
asig = K35_hpf(asig, expseg:k(10000, p3, 30), 9.9, 0, 1)
asig *= 0.25
asig = limit(asig, -1.0, 1.0)

outc(asig, asig)

endin

;; beat instruments

instr ms20_drum

ipch = cps2pch(p4, 12)
iamp = ampdbfs(p5)
aenv = expseg:a(10000, 0.05, ipch, p3 - .05, ipch)

asig = rand:a(-1.0, 1.0)
asig = K35_hpf(asig, 60, 7, 1, 1)
asig = K35_lpf(asig, aenv, 9.8, 1, 1)

asig = tanh(asig * 16)

asig *= expon(iamp, p3, 0.0001)

outc(asig, asig)

endin

instr ms20_bass
```

```

    ipch = cps2pch(p4, 12)
    iamp = ampdbfs(p5)
    aenv = expseg(1000, 0.1, ipch * 2, p3 - .05, ipch * 2)

    asig = vco2(1.0, ipch)
    asig = K35_hpf(asig, ipch, 5, 0, 1)
    asig = K35_lpf(asig, aenv, 8, 0, 1)

    asig *= expon:a(iamp, p3, 0.0001) * 0.8

    outc(asig, asig)
  endin

;; perf code

gktempo init 122

opcode beat_dur,i,0
  xout 60 / i(gktempo)
endop

instr bass_player
  idur = beat_dur() / int(random(1,3))
  ipch = 6.00 + int(random(1,3)) + int(random(1,3)) / 100

  schedule("ms20_bass", 0, idur, ipch, -11)

  if(p2 < 37.5) then
    schedule("bass_player", idur, 0.1)
  endif
  turnoff
endin

instr beat_player
  istep_total = p4
  istep = istep_total % 16

  if(istep % 4 == 0) then
    ipch = ((istep_total % 128) < 112) ? 4.00 : 8.00
    iamp = (istep == 0) ? -9 : -12
    schedule("ms20_drum", 0, 0.5, ipch, iamp)
  endif

  schedule("ms20_drum", 0, 0.125, 14.00,
    (istep % 4 == 0) ? -12 : -18)

  if(p2 < 37.5) then
    schedule("beat_player", beat_dur() / 4, 0.1, istep_total + 1)
  endif
  turnoff
endin

;; start play of beats

instr start_beats
  schedule("beat_player", 0, 0.1, 0)
  schedule("bass_player", 0, 0.1)
endin

</CsInstruments>
; =====
<CsScore>
i1 0 5.0
i2 5 5.0
i3 10 5.0
i4 15 5.0

```

```
i "start_beats" 22 0.5 0
```

```
</CsScore>
```

```
</CsoundSynthesizer>
```

## References

This filter is based on the work of Will Pirkle that employs Vadim Zavalishin's work with bilinear transforms to create topology-preserving transform (TPT) implementations of analog filters.

1. Pirkle, Will. Designing Software Synthesizer Plug-ins in C++: For RackAFX, VST3, and Audio Units. CRC Press, 2014.
2. Pirkle, Will. AN-7A: Virtual Analog (VA) Korg35 Highpass Filter v2.0 Simplified. 2013.
3. Zavalishin, Vadim. "The Art of VA filter design." Native Instruments, 2012.

## Credits

Author: Steven Yi

April 2017

New in Csound 6.09.0

# K35\_lpf

K35\_lpf — Zero-delay feedback implementation of Korg35 resonant low-pass filter.

## Description

Zero-delay feedback implementation of Korg35 resonant low-pass filter. This filter design is found in the Korg MS10, early MS20, and Monotron series.

## Syntax

```
asig k35_lpf ain, xcf, xQ [, inlp, isaturation, istor]
```

## Initialization

*istor* --initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*asig* -- output signal.

*ain* -- input signal.

*xcf* -- filter cutoff frequency (i-, k-, or a-rate).

*xQ* -- filter Q value (i-, k-, or a-rate). Range 1.0-10.0 (clamped by opcode). Self-oscillation occurs at 10.0.

*knlp* (optional, default=0) -- Non-linear processing method. 0 = no processing, 1 = non-linear processing. Method 1 uses  $\tanh(ksaturation * input)$ . Enabling NLP may increase the overall output of filter above unity and should be compensated for outside of the filter.

*ksaturation* (optional, default=1) -- saturation amount to use for non-linear processing. Values > 1 increase the steepness of the NLP curve.

## Examples

Here is an example of the K35\_lpf opcode. It uses the file *k35.csd* [examples/k35.csd].

### Example 450. Example of the K35\_lpf opcode.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
; =====
<CsInstruments>

sr = 48000
ksmps = 1
nchnls = 2
odbfs = 1

;; test instruments to demo filter cutoff sweep with high resonance
```

```
instr 1

asig = vco2(0.5, cps2pch(6.00, 12))
asig = K35_lpf(asig, expseg:a(10000, p3, 30), 9.9, 0, 1)
asig *= 0.25
asig = limit(asig, -1.0, 1.0)

outc(asig, asig)

endin

instr 2

asig = vco2(0.5, cps2pch(6.00, 12))
asig = K35_lpf(asig, expseg:k(10000, p3, 30), 9.9, 0, 1)
asig *= 0.25
asig = limit(asig, -1.0, 1.0)

outc(asig, asig)

endin

instr 3

asig = vco2(0.5, cps2pch(6.00, 12))
asig = K35_hpf(asig, expseg:a(10000, p3, 30), 9.9, 0, 1)
asig *= 0.25
asig = limit(asig, -1.0, 1.0)

outc(asig, asig)

endin

instr 4

asig = vco2(0.5, cps2pch(6.00, 12))
asig = K35_hpf(asig, expseg:k(10000, p3, 30), 9.9, 0, 1)
asig *= 0.25
asig = limit(asig, -1.0, 1.0)

outc(asig, asig)

endin

;; beat instruments

instr ms20_drum

ipch = cps2pch(p4, 12)
iamp = ampdbfs(p5)
aenv = expseg:a(10000, 0.05, ipch, p3 - .05, ipch)

asig = rand:a(-1.0, 1.0)
asig = K35_hpf(asig, 60, 7, 1, 1)
asig = K35_lpf(asig, aenv, 9.8, 1, 1)

asig = tanh(asig * 16)

asig *= expon(iamp, p3, 0.0001)

outc(asig, asig)

endin

instr ms20_bass
```

```

ipch = cps2pch(p4, 12)
iamp = ampdbfs(p5)
aenv = expseg(1000, 0.1, ipch * 2, p3 - .05, ipch * 2)

asig = vco2(1.0, ipch)
asig = K35_hpf(asig, ipch, 5, 0, 1)
asig = K35_lpf(asig, aenv, 8, 0, 1)

asig *= expon:a(iamp, p3, 0.0001) * 0.8

outc(asig, asig)
endin

;; perf code

gktempo init 122

opcode beat_dur,i,0
  xout 60 / i(gktempo)
endop

instr bass_player
  idur = beat_dur() / int(random(1,3))
  ipch = 6.00 + int(random(1,3)) + int(random(1,3)) / 100

  schedule("ms20_bass", 0, idur, ipch, -11)

  if(p2 < 37.5) then
    schedule("bass_player", idur, 0.1)
  endif
  turnoff
endin

instr beat_player
  istep_total = p4
  istep = istep_total % 16

  if(istep % 4 == 0) then
    ipch = ((istep_total % 128) < 112) ? 4.00 : 8.00
    iamp = (istep == 0) ? -9 : -12
    schedule("ms20_drum", 0, 0.5, ipch, iamp)
  endif

  schedule("ms20_drum", 0, 0.125, 14.00,
    (istep % 4 == 0) ? -12 : -18)

  if(p2 < 37.5) then
    schedule("beat_player", beat_dur() / 4, 0.1, istep_total + 1)
  endif
  turnoff
endin

;; start play of beats

instr start_beats
  schedule("beat_player", 0, 0.1, 0)
  schedule("bass_player", 0, 0.1)
endin

</CsInstruments>
; =====
<CsScore>
i1 0 5.0
i2 5 5.0
i3 10 5.0
i4 15 5.0

```

```
i "start_beats" 22 0.5 0  
  
</CsScore>  
</CsoundSynthesizer>
```

## References

This filter is based on the work of Will Pirkle that employs Vadim Zavalishin's work with bilinear transforms to create topology-preserving transform (TPT) implementations of analog filters.

1. Pirkle, Will. Designing Software Synthesizer Plug-ins in C++: For RackAFX, VST3, and Audio Units. CRC Press, 2014.
2. Pirkle, Will. AN-5: Virtual Analog (VA) Korg35 Lowpass Filter v3.5. 2013.
3. Zavalishin, Vadim. "The Art of VA filter design." Native Instruments, 2012.

## Credits

Author: Steven Yi  
April 2017

New in Csound 6.09.0

# kgoto

kgoto — Transfer control during the performance-time passes.

## Description

During the p-time passes only, unconditionally transfer control to the statement labeled by *label*.

## Syntax

```
kgoto label
```

where *label* is in the same instrument block and is not an expression.

## Examples

Here is an example of the kgoto opcode. It uses the file *kgoto.csd* [examples/kgoto.csd].

### Example 451. Example of the kgoto opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o kgoto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Change kval linearly from 0 to 2 over
; the period set by the third p-field.
kval line 0, p3, 2

; If kval is greater than or equal to 1 then play the high note.
; If not then play the low note.
if (kval >= 1) kgoto highnote
    kgoto lownote

highnote:
    kfreq = 880
    goto playit

lownote:
    kfreq = 440
    goto playit
```

```
playit:
; Print the values of kval and kfreq.
printks "kval = %f, kfreq = %f\\n", 1, kval, kfreq

a1 oscil 10000, kfreq, 1
out a1
endin

</CsInstruments>
<CsScore>

; Table #1: a simple sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
kval = 0.000000, kfreq = 440.000000
kval = 0.999732, kfreq = 440.000000
kval = 1.999639, kfreq = 880.000000
```

## See Also

*cggoto, cigoto, ckgoto, goto, if, igoto, tigoto, timeout*

## Credits

Example written by Kevin Conder.

# kr

kr — Sets the control rate.

## Description

These statements are global value *assignments*, made at the beginning of an orchestra, before any instrument block is defined. Their function is to set certain *reserved symbol variables* that are required for performance. Once set, these reserved symbols can be used in expressions anywhere in the orchestra.

## Syntax

```
kr = iarg
```

## Initialization

*kr* = (optional) -- set control rate to *iarg* samples per second. The default value is 4410.

In addition, any *global variable* can be initialized by an *init-time assignment* anywhere before the first *instr statement*. All of the above assignments are run as instrument 0 (i-pass only) at the start of real performance.

Beginning with Csound version 3.46, *kr* can be omitted. Csound will use the default values, or calculate *kr* from defined *ksmps* and *sr*. It is usually better to just specify *ksmps* and *sr* and let csound calculate *kr*.

## Examples

```
sr = 10000
kr = 500
ksmps = 20
gil = sr/2.
ga init 0
itranspose = octpch(.01)
```

## See Also

*ksmps*, *nchnls*, *nchnls\_i*, *sr*

# ksmps

ksmps — Sets the number of samples in a control period.

## Description

These statements are global value *assignments*, made at the beginning of an orchestra, before any instrument block is defined. Their function is to set certain *reserved symbol variables* that are required for performance. Once set, these reserved symbols can be used in expressions anywhere in the orchestra.

## Syntax

```
ksmps = iarg
```

## Initialization

*ksmps* = (optional) -- set the number of samples in a control period. This value must equal *sr*/*kr*. The default value is 10.

In addition, any *global variable* can be initialized by an *init-time assignment* anywhere before the first *instr statement*. All of the above assignments are run as instrument 0 (i-pass only) at the start of real performance.

Beginning with Csound version 3.46, either *ksmps* may be omitted. Csound will attempt to calculate the omitted value from the specified *sr* and *kr* values, but it should evaluate to an integer.



### Warning

ksmps must be an integer value.

## Examples

```
sr = 10000
kr = 500
ksmps = 20
gil = sr/2.
ga init 0
itranspose = octpch(.01)
```

## See Also

*kr*, *nchnls*, *nchnls\_i*, *sr*

## Credits

Thanks to a note from Gabriel Maldonado, added a warning about integer values.

# lenarray

lenarray — Evaluates the size or shape length of an array.

## Description

Evaluates the size or number of dimensions of an array.

## Syntax

```
ir lenarray karray[, iwhich]
kr lenarray karray[, iwhich]
```

## Initialisation

*karray* -- The array that is being questioned. It can be of any dimension.

*iwhich* -- selects which dimension to evaluate the size. If zero or negative it selects the number of dimensions. It defaults to 1, as used in a vector.

## Performance

*kr* -- length of vector.

*karray* -- array to query.

If the dimension requested is larger than the actual array, or the array is not initialised the value -1 is returned.

## Examples

Here is an example of the lenarray opcode. It uses the file *lenarray.csd* [examples/lenarray.csd].

### Example 452. Example of the lenarray opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-n -m128 ;no sound output, reduced messages
</CsOptions>
<CsInstruments>
;example by joachim heintz
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

instr 1 ;simple example
kArr[] fillarray 1, 2, 3, 4 ;fill array manually
printks "Length of kArr = %d\n\n", 0, lenarray(kArr) ;print out its length
turnoff ;only do this in the first k-cycle
```

```
    endin

    instr 2 ;random array length
iNumEls random 1, 11 ;create random number between 1 and 10
kArr[] init int(iNumEls) ;create array of this length
    printks "Random length of kArr = %d\n", 0, lenarray(kArr) ;print out
    turnoff
    endin

    instr 3 ;fill random array length with random elements
iNumEls random 1, 11 ;create random number between 1 and 10
kArr[] init int(iNumEls) ;create array of this length
    printks "Random length of kArr = %d\n", 0, lenarray(kArr) ;print out

;fill
kIndx = 0 ;initialize index
    until kIndx == lenarray(kArr) do
kArr[kIndx] rnd31 10, 0 ;set element to random value -10...10
kIndx += 1 ;increase index
    od

;print
kIndx = 0 ;initialize index
    until kIndx == lenarray(kArr) do
printf("kArr[%d] = %f\n", kIndx+1, kIndx, kArr[kIndx])
kIndx += 1 ;increase index
    od

    turnoff
    endin
</CsInstruments>
<CsScore>
i 1 0 .1
i 2 0 .1
i 2 0 .1
i 3 0 .1
</CsScore>
</CsoundSynthesizer>
```

## See Also

Vectorial opcodes

## Credits

Author: John ffitth  
Codemist Ltd  
2013

New in version 6.00

Extended to multi-dimensions in 6.04

# lfo

lfo — A low frequency oscillator of various shapes.

## Description

A low frequency oscillator of various shapes.

## Syntax

```
kres lfo kamp, kcps [, itype]
```

```
ares lfo kamp, kcps [, itype]
```

## Initialization

*itype* (optional, default=0) -- determine the waveform of the oscillator. Default is 0.

- *itype* = 0 - sine
- *itype* = 1 - triangles
- *itype* = 2 - square (bipolar)
- *itype* = 3 - square (unipolar)
- *itype* = 4 - saw-tooth
- *itype* = 5 - saw-tooth(down)

The sine wave is implemented as a 4096 table and linear interpolation. The others are calculated.

## Performance

*kamp* -- amplitude of output

*kcps* -- frequency of oscillator

## Examples

Here is an example of the lfo opcode. It uses the file *lfo.csd* [examples/lfo.csd].

### Example 453. Example of the lfo opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
```

```
; -o lfo.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
Odbfs = 1

instr 1

kcps = 5
itype = p4 ;lfo type

klfo line 0, p3, 20
al lfo klfo, kcps, itype
asig poscil .5, 220+al, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
; sine wave.
f 1 0 32768 10 1

i 1 0 3 0 ;lfo = sine
i 1 + 3 2 ;lfo = square
i 1 + 3 5 ;lfo = saw-tooth down
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
November 1998

New in Csound version 3.491

# limit

limit — Sets the lower and upper limits of the value it processes.

## Description

Sets the lower and upper limits of the value it processes.

## Syntax

```
ares limit asig, klow, khigh
ires limit isig, ilow, ihigh
kres limit ksig, klow, khigh
ires[] limit isig[], ilow, ihigh
kres[] limit ksig[], klow, khigh
```

## Initialization

*isig* -- input signal

*ilow* -- low threshold

*ihigh* -- high threshold

## Performance

*xsig* -- input signal

*klow* -- low threshold

*khigh* -- high threshold

*limit* sets the lower and upper limits on the *xsig* value it processes. If *xhigh* is lower than *xlow*, then the output will be the average of the two - it will not be affected by *xsig*.

This opcode is useful in several situations, such as table indexing or for clipping and modeling a-rate, i-rate or k-rate signals.

## Examples

Here is an example of the limit opcode. It uses the file *limit.csd* [examples/limit.csd].

### Example 454. Example of the limit opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
```

```

; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o limit.wav -W   ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
Odbfs = 1
nchnls = 2

instr   1 ; Limit / Mirror / Wrap

igain   = p4   ;gain
ilevl1  = p5   ; + level
ilevl2  = p6   ; - level
imode   = p7   ;1 = limit, 2 = mirror, 3 = wrap

ain     soundin "fox.wav"
ain     = ain*igain

if      imode = 1 goto limit
if      imode = 2 goto mirror

asig    wrap ain, ilevl2, ilevl1
goto    outsignal

limit:
asig    limit ain, ilevl2, ilevl1
goto    outsignal

mirror:
asig    mirror ain, ilevl2, ilevl1
outsignal:

outs    asig*.5, asig*.5   ;mind your speakers

endin

</CsInstruments>
<CsScore>

;          Gain +Levl -Levl Mode
i1 0 3    4.00 .25 -1.00 1 ;limit
i1 4 3    4.00 .25 -1.00 2 ;mirror
i1 8 3    4.00 .25 -1.00 3 ;wrap
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*mirror, wrap*

## Credits

Author: Robin Whittle  
Australia

New in Csound version 3.46

# limit1

limit1 — Limiting function

## Description

Limits the value of an argument to the range [0,1].

## Syntax

```
ires[] limit1 iarg
kres[] limit1 karg
```

## Initialization

*iarg[]* -- the argument.

## Performance

*karg[]* -- the argument.

## Examples

Here is an example of the limit1 opcode. It uses the file *limit1.csd* [examples/limit1.csd].

### Example 455. Example of the limit1 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>

instr 1

iArg1[] fillarray 1,2,3
iRes[] limit1 iArg1/2.5
ik init 0

while ik < lenarray(iRes) do
  print iRes[ik]
  ik += 1
od

endin

</CsInstruments>
<CsScore>
i1 0 0
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
2017

# line

line — Trace a straight line between specified points.

## Description

Trace a straight line between specified points.

## Syntax

```
ares line ia, idur, ib
kres line ia, idur, ib
```

## Initialization

*ia* -- starting value.

*ib* -- value after *idur* seconds.

*idur* -- duration in seconds of segment. A zero or negative value will cause all initialization to be skipped.

## Performance

*line* generates control or audio signals whose values move linearly from an initial value to a final one.



### Note

A common error with this opcode is to assume that the value of *ib* is held after the time *idur*. *line* does not automatically end or stop at the end of the duration given. If your note length is longer than *idur* seconds, *kres* (or *ares*) will not come to rest at *ib*, but will instead continue to rise or fall with the same rate. If a rise (or fall) and then hold is required that the *linseg* opcode should be considered instead.

## Examples

Here is an example of the line opcode. It uses the file *line.csd* [examples/line.csd].

### Example 456. Example of the line opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o line.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kp = p6
;choose between expon or line
if (kp == 0) then
  kpitch expon p4, p3, p5
elseif (kp == 1) then
  kpitch line p4, p3, p5
endif

asig vco2 .6, kpitch
outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 2 300 600 0 ;if p6=0 then expon is used
i 1 3 2 300 600 1 ;if p6=1 then line is used
i 1 6 2 600 1200 0
i 1 9 2 600 1200 1
i 1 12 2 1200 2400 0
i 1 15 2 1200 2400 1
i 1 18 2 2400 30 0
i 1 21 2 2400 30 1
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*expon, expseg, expsegr, linseg, linsegr*

## Credits

# linen

`linen` — Applies a straight line rise and decay pattern to an input amp signal.

## Description

`linen` -- apply a straight line rise and decay pattern to an input amp signal.

## Syntax

```
ares linen xamp, irise, idur, idec
```

```
kres linen kamp, irise, idur, idec
```

## Initialization

`irise` -- rise time in seconds. A zero or negative value signifies no rise modification.

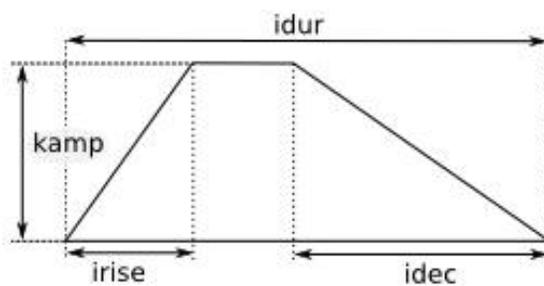
`idur` -- overall duration in seconds. A zero or negative value will cause initialization to be skipped.

`idec` -- decay time in seconds. Zero means no decay. An `idec > idur` will cause a truncated decay.

## Performance

`kamp`, `xamp` -- input amplitude signal.

Rise modifications are applied for the first `irise` seconds, and decay from time `idur - idec`. If these periods are separated in time there will be a steady state during which `amp` will be unmodified. If `linen` rise and decay periods overlap then both modifications will be in effect for that time. If the overall duration `idur` is exceeded in performance, the final decay will continue on in the same direction, going negative.



Envelope generated by the `linen` opcode



### Note

A common error with this opcode is to assume that the value of 0 is the held after the envelope has finished at `idur`. `linen` does not automatically end or stop at the end of the duration given. If your note length is longer than `idur` seconds, `kres` (or `ares`) will not come to rest at 0, but will instead continue to fall with the same rate. If a decay and then hold is required then the `linseg` opcode should be considered instead.

## Examples

Here is an example of the `linen` opcode. It uses the file `linen.csd` [examples/linen.csd].

### Example 457. Example of the `linen` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o linen.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
; p4=amp
; p5=freq
; p6=attack time
; p7=release time
ares linen p4, p6, p3, p7
asig poscil ares, p5, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
f1 0 4096 10 1 ; sine wave

;ins strt dur amp freq attack release
i1 0 1 .5 440 0.5 0.7
i1 1.5 1 .2 440 0.9 0.1
i1 3 1 .2 880 0.02 0.99
i1 4.5 1 .2 880 0.7 0.01
i1 6 3 .7 220 0.5 0.5
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*envlpx, envlpxr, linenr*

# linenr

linenr — The *linen* opcode extended with a final release segment.

## Description

*linenr* -- same as *linen* except that the final segment is entered only on sensing a MIDI note release. The note is then extended by the decay time.

## Syntax

```
ares linenr xamp, irise, idec, iatdec
```

```
kres linenr kamp, irise, idec, iatdec
```

## Initialization

*irise* -- rise time in seconds. A zero or negative value signifies no rise modification.

*idec* -- decay time in seconds. Zero means no decay. An *idec* > *idur* will cause a truncated decay.

*iatdec* -- attenuation factor by which the closing steady state value is reduced exponentially over the decay period. This value must be positive and is normally of the order of .01. A large or excessively small value is apt to produce a cutoff which is audible. A zero or negative value is illegal.

## Performance

*kamp*, *xamp* -- input amplitude signal.

*linenr* is unique within Csound in containing a *note-off sensor* and *release time extender*. When it senses either a score event termination or a MIDI noteoff, it will immediately extend the performance time of the current instrument by *idec* seconds, then execute an exponential decay towards the factor *iatdec*. For two or more units in an instrument, extension is by the greatest *idec*.

You can use other pre-made envelopes which start a release segment upon receiving a note off message, like *linsegr* and *expsegr*, or you can construct more complex envelopes using *xtratim* and *release*. Note that you don't need to use *xtratim* if you are using *linenr*, since the time is extended automatically.

These “r” units can also be modified by MIDI noteoff velocities (see *veloffs*).

## Examples

Here is an example of the *linenr* opcode. It uses the file *linenr.csd* [examples/*linenr.csd*].

### Example 458. Example of the *linenr* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc          -M0 ;;RT audio I/O with MIDI in
</CsOptions>
```

```

<CsInstruments>

; Example by Jonathan Murphy and Charles Gran 2007
sr      = 44100
ksmps   = 10
nchnls  = 2

; new, and important. Make sure that midi note events are only
; received by instruments that actually need them.

; turn default midi routing off
massign 0, 0
; route note events on channel 1 to instr 1
massign 1, 1

; Define your midi controllers
#define C1 #21#
#define C2 #22#
#define C3 #23#

; Initialize MIDI controllers
initc7   1, 21, 0.5    ;delay send
initc7   1, 22, 0.5    ;delay: time to zero
initc7   1, 23, 0.5    ;delay: rate

gaosc    init        0

; Define an opcode to "smooth" the MIDI controller signal
opcode smooth, k, k
kin      xin
kport   linseg      0, 0.0001, 0.01, 1, 0.01
kin      portk      kin, kport
kout    xout        kin
endop

instr 1
; Generate a sine wave at the frequency of the MIDI note that triggered the instrument
ifgc    cpsmidi
iamp    ampmidi      10000
aenv    linenr      iamp, .01, .1, .01    ;envelope
a1      oscil        aenv, ifgc, 1
; All sound goes to the global variable gaosc
gaosc   = gaosc + a1
endin

instr 198 ; ECHO
kcbsnd  ctrl7       1, 21, 0, 1    ;delay send
ktime   ctrl7       1, 22, 0.01, 6    ;time loop fades out
kloop   ctrl7       1, 23, 0.01, 1    ;loop speed
; Receive MIDI controller values and then smooth them
kcbsnd  smooth      kcbsnd
ktime   smooth      ktime
kloop   smooth      kloop

imaxlpt = 1          ;max loop time
; Create a variable reverberation (delay) of the gaosc signal
acomb   vcomb       gaosc, ktime, kloop, imaxlpt, 1
aout    = (acomb * kcbsnd) + gaosc * (1 - kcbsnd)
outs    aout, aout
gaosc   = 0
endin

</CsInstruments>

<CsScore>
f1 0 16384 10 1
i198 0 10000

```

```
e  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*linsegr, expsegr, envlpxr, mxadsr, madsr, envlpx, linen, xtratim*

# lineto

lineto — Generate glissandos starting from a control signal.

## Description

Generate glissandos starting from a control signal.

## Syntax

```
kres lineto ksig, ktime
```

## Performance

*kres* -- Output signal.

*ksig* -- Input signal.

*ktime* -- Time length of glissando in seconds.

*lineto* adds glissando (i.e. straight lines) to a stepped input signal (for example, produced by *randh* or *lpshold*). It generates a straight line starting from previous step value, reaching the new step value in *ktime* seconds. When the new step value is reached, such value is held until a new step occurs. Be sure that *ktime* argument value is smaller than the time elapsed between two consecutive steps of the original signal, otherwise discontinuities will occur in output signal.

When used together with the output of *lpshold* it emulates the glissando effect of old analog sequencers.



### Note

No new value for *ksig* or *ktime* will have effect until the previous *ktime* has elapsed.

## Examples

Here is an example of the *lineto* opcode. It uses the file *lineto.csd* [examples/lineto.csd].

### Example 459. Example of the *lineto* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o lineto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
```

```
nchnls = 2
odbfs = 1

giSine ftgen 0, 0, 2^10, 10, 1

instr 1

kfreq   randh   1000, 20, 2, 1, 2000 ;generates ten random number between 100 and 300 per second
kpan    randh   .5, 1, 2, 1, .5    ;panning between 0 and 1
kp      lineto  kpan, .5          ;smoothing pan transition
aout    poscil  .4, kfreq, giSine
aL, aR  pan2    aout, kp
outs    outs    aL, aR

endin
</CsInstruments>
</CsScore>

i 1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*tlineto*

## Credits

Author: Gabriel Maldonado

New in Version 4.13

# linlin

linlin — Linear to linear interpolation

## Description

Maps a linear range of values to another linear range of values.

## Syntax

```
kout linlin kin, ksrclo, ksrchi, kdstlo, kdsthi
```

## Performance

*kin* -- Input signal

*ksrclo* -- Lower limit of input range

*ksrchi* -- Higher limit of input range

*kdstlo* -- Lower limit of output range

*kdsthi* -- Higher limit of output range

$$\text{out} = (\text{in} - \text{srclo}) / (\text{srchi} - \text{srclo}) * (\text{dsthi} - \text{dstlo}) + \text{dstlo}$$

## Examples

Here is an example of the linlin opcode. It uses the file *linlin.csd* [examples/linlin.csd].

### Example 460. Example of the linlin opcode.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 128
nchnls = 2
0dbfs = 1.0

; Map a value within the range 1-3 to the range 0-10.

instr 1
  kx line 1, p3, 3
  ky linlin kx, 1, 3, 0, 10
  printks "kx: %f   ky: %f \n", 1/kr, kx, ky
endin

</CsInstruments>
<CsScore>
i 1 0 1
```

```
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*scale, ntrpol, bpf*

## Credits

By: Eduardo Moguillansky 2017

New in version 6.09

# link\_beat\_force

`link_beat_force` — Forces the global network Ableton Link session to adopt a specific beat number and time.

## Description

Forces the global network Ableton Link session to adopt a specific beat number and time, like a conductor stopping an orchestra and immediately starting it again.

## Syntax

```
link_beat_force i_peer, k_beat [, k_at_time_seconds [, k_quantum ]]
```

## Initialization

*i\_peer* -- The Ableton Link peer object.

## Performance

*k\_beat* -- The number requested for the beat.

*k\_at\_time\_seconds* -- The time requested for the beat; the default value is the current time.

*k\_quantum* -- The local quantum of the beat; the default value is 1.

## See Also

*Ableton Link Opcodes*

## Credits

By: Michael Gogins 2017

## link\_beat\_get

`link_beat_get` — Returns the beat, phase with respect to the local quantum, and current time for the session.

### Description

Returns the beat number, phase of the beat with respect to the local quantum of the beat, and current time for the global network Ableton Link session.

### Syntax

```
k_beat_number, k_phase, k_current_time_seconds link_beat_get i_peer [, k_quantum]
```

### Initialization

*i\_peer* -- The Ableton Link peer object.

### Performance

*k\_quantum* -- The local quantum of the beat; the default value is 1,

*k\_beat\_number* -- The number of the current beat.

*k\_phase* -- The current phase of the beat with respect to the local quantum of the beat.

*k\_current\_time\_seconds* -- The current time with respect to the time this peer was enabled.

Consider using `link_metro` if you need a periodic trigger driven by the session beat.

### See Also

*Ableton Link Opcodes*

### Credits

By: Michael Gogins 2017

# link\_beat\_request

`link_beat_request` — Requests the global network Ableton Link session to adopt a specific beat number and time.

## Description

Requests the global network Ableton Link session to adopt a specific beat number and time.

## Syntax

```
link_beat_request i_peer, k_beat [, k_at_time_seconds [, k_quantum ]]
```

## Initialization

*i\_peer* -- The Ableton Link peer object.

## Performance

*k\_beat* -- The number requested for the beat.

*k\_at\_time\_seconds* -- The time requested for the beat; the default value is the current time.

*k\_quantum* -- The local quantum of the beat; the default value is 1.

## See Also

*Ableton Link Opcodes*

## Credits

By: Michael Gogins 2017

# link\_create

`link_create` — Creates a peer in an Ableton Link network session.

## Description

Creates a peer in an Ableton Link network session. The first peer in a session determines the initial tempo of the session. The value returned must be passed as the first parameter to all subsequent Ableton Link opcode calls for this peer.

## Syntax

```
i_peer link_create [i_bpm]
```

## Initialization

`i_bpm` -- Initial tempo of the session, in beats per minute. Has no effect unless this is the first peer in the session. The default value is 60.

`i_peer` -- The Ableton Link peer object returned for use with the other Ableton Link opcodes.

## Examples

Here is an example of the `link_create` opcode. It uses the file `ableton_link.csd` [examples/ableton\_link.csd].

### Example 461. Example of the `link_create` opcode.

```
<CsoundSynthesizer>
<CsLicense>
Run the Ableton Link "LinkHut" example application, or some other
Ableton Link peer, while you run this CSD to see what happens.
</CsLicense>
<CsOptions>
-m0 -d -odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 10
nchnls = 2

alwayson "LinkMonitor"

gi_peer link_create 72
gk_beat init 0

instr Beep
asignal oscils 20000, p4, 0
outs asignal, asignal
endin

instr TempoChange
link_tempo_set gi_peer, 80
endin

instr LinkEnable
```

```
i_enabled = p4
link_enable gi_peer, i_enabled
endin

instr LinkMonitor
i_kperiod_seconds = ksmps / sr
printf_i "kperiod: %9.6f seconds.\n", 1, i_kperiod_seconds
printf_i "gi_peer: %g\n", 1, gi_peer
link_enable gi_peer, 1
k_trigger, gk_beat, k_phase, k_time link_metro gi_peer, 1
k_peers link_peers gi_peer
k_tempo link_tempo_get gi_peer
k_enabled link_is_enabled gi_peer
k_hz = 1000
if floor(gk_beat % 4) == 0 then
k_hz = 3000
else
k_hz = 2000
endif
schedkwhen k_trigger, 0, 1, "Beep", 0, 0.01, k_hz
printf "LinkMonitor: gi_peer: %g k_enabled: %9.4f k_trigger: %9.4f beat: %9.4f k_phase: %9.4f time: %9.4f\n", 1, k_enabled, k_trigger, gk_beat, k_phase, k_time
endin
</CsInstruments>
<CsScore>
f 0 360
i "TempoChange" 10 80
i "LinkEnable" 20 1 0
i "LinkEnable" 30 1 1
</CsScore>
</CsoundSynthesizer>
```

## See Also

See the section *Ableton Link Opcodes* for more information.

## Credits

By: Michael Gogins 2017

# link\_enable

link\_enable — Enable or disable synchronization with the Ableton Link session.

## Description

Enable or disable synchronization with the global network Ableton Link session tempo and beat.

## Syntax

```
ableton_link_enable i_peer [, k_enable]
```

## Initialization

*i\_peer* -- The Ableton Link peer object.

## Performance

*k\_enable* -- Enables (non-zero value) or disables (zero value) synchronization of this peer with the Ableton Link session.

## Examples

Here is an example of the link\_enable opcode. It uses the file *ableton\_link.csd* [examples/ableton\_link.csd].

### Example 462. Example of the link\_enable opcode.

```
<CsoundSynthesizer>
<CsLicense>
Run the Ableton Link "LinkHut" example application, or some other
Ableton Link peer, while you run this CSD to see what happens.
</CsLicense>
<CsOptions>
-m0 -d -odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 10
nchnls = 2

alwayson "LinkMonitor"

gi_peer link_create 72
gk_beat init 0

instr Beep
asignal oscils 20000, p4, 0
outs asignal, asignal
endin

instr TempoChange
link_tempo_set gi_peer, 80
endin
```

```
instr LinkEnable
i_enabled = p4
link_enable gi_peer, i_enabled
endin

instr LinkMonitor
i_kperiod_seconds = ksmpr / sr
printf_i "kperiod: %9.6f seconds.\n", 1, i_kperiod_seconds
printf_i "gi_peer: %g\n", 1, gi_peer
link_enable gi_peer, 1
k_trigger, gk_beat, k_phase, k_time link_metro gi_peer, 1
k_peers link_peers gi_peer
k_tempo link_tempo_get gi_peer
k_enabled link_is_enabled gi_peer
k_hz = 1000
if floor(gk_beat % 4) == 0 then
k_hz = 3000
else
k_hz = 2000
endif
schedkwhen k_trigger, 0, 1, "Beep", 0, 0.01, k_hz
printf "LinkMonitor: gi_peer: %g k_enabled: %9.4f k_trigger: %9.4f beat: %9.4f k_phase: %9.4f time: %9.4f\n", 1, k_enabled, k_trigger, gk_beat, k_phase, k_time
endin
</CsInstruments>
<CsScore>
f 0 360
i "TempoChange" 10 80
i "LinkEnable" 20 1 0
i "LinkEnable" 30 1 1
</CsScore>
</CsoundSynthesizer>
```

## See Also

See the section *Ableton Link Opcodes* for more information.

## Credits

By: Michael Gogins 2017

# link\_is\_enabled

`link_is_enabled` — Returns whether or not this peer is synchronized with the global network Ableton Link session.

## Description

Returns whether or not the beat and time of his peer are synchronized with the global network Ableton Link session.

## Syntax

```
k_is_enabled link_is_enabled i_peer
```

## Initialization

`i_peer` -- The Ableton Link peer object.

## Performance

`k_is_enabled` -- Returns 1 if this peer is enabled, or 0 otherwise.

## Examples

Here is an example of the `link_is_enabled` opcode. It uses the file `ableton_link.csd` [examples/ableton\_link.csd].

### Example 463. Example of the `link_is_enabled` opcode.

```
<CsoundSynthesizer>
<CsLicense>
Run the Ableton Link "LinkHut" example application, or some other
Ableton Link peer, while you run this CSD to see what happens.
</CsLicense>
<CsOptions>
-m0 -d -odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 10
nchnls = 2

alwayson "LinkMonitor"

gi_peer link_create 72
gk_beat init 0

instr Beep
asignal oscils 20000, p4, 0
outs asignal, asignal
endin

instr TempoChange
link_tempo_set gi_peer, 80
```

```
endin

instr LinkEnable
i_enabled = p4
link_enable gi_peer, i_enabled
endin

instr LinkMonitor
i_kperiod_seconds = ksmps / sr
printf_i "kperiod: %9.6f seconds.\n", 1, i_kperiod_seconds
printf_i "gi_peer: %g\n", 1, gi_peer
link_enable gi_peer, 1
k_trigger, gk_beat, k_phase, k_time link_metro gi_peer, 1
k_peers link_peers gi_peer
k_tempo link_tempo_get gi_peer
k_enabled link_is_enabled gi_peer
k_hz = 1000
if floor(gk_beat % 4) == 0 then
k_hz = 3000
else
k_hz = 2000
endif
schedkwhen k_trigger, 0, 1, "Beep", 0, 0.01, k_hz
printf "LinkMonitor: gi_peer: %g k_enabled: %9.4f k_trigger: %9.4f beat: %9.4f k_phase: %9.4f time: %9.4f\n", 1, gi_peer, k_enabled, k_trigger, gk_beat, k_phase, k_time
endin
</CsInstruments>
<CsScore>
f 0 360
i "TempoChange" 10 80
i "LinkEnable" 20 1 0
i "LinkEnable" 30 1 1
</CsScore>
</CsoundSynthesizer>
```

## See Also

See the section *Ableton Link Opcodes* for more information.

## Credits

By: Michael Gogins 2017

# link\_metro

`link_metro` — Returns a trigger that is 1 on the beat and 0 otherwise along with beat, phase, and time for this session of Ableton Link.

## Description

Returns a trigger that is 1 on the beat and 0 otherwise along with the beat, phase, and current time of Ableton Link for this session for a given quantum.

## Syntax

```
k_trigger, k_beat, k_phase, k_current_time_seconds link_metro i_peer [, k_quantum]
```

## Initialization

*i\_peer* -- The Ableton Link peer object.

## Performance

*k\_trigger* -- Trigger, 1 at the instant of the first beat in each local quantum of the beat, and 0 otherwise.

*k\_beat* -- The number of the beat, counting from when this peer is enabled.

*k\_phase* -- The phase of the beat with respect to the quantum.

*k\_current\_time\_seconds* -- The time of the beat, counting from when this peer is enabled.

*k\_quantum* -- The local quantum of the beat, which may be a multiple or fraction of a beat. The default value is 1.

This opcode permits the use of an Ableton Link session beat to trigger events in Csound. It also can be used to obtain the session time and beat, as well as the phase with respect to the local quantum of the beat. For example, a quantum of 4 can imply 4/4 time, or a quantum of 0.25 will subdivide the beat by 4.

## Examples

Here is an example of the `link_metro` opcode. It uses the file `ableton_link.csd` [examples/ableton\_link.csd].

### Example 464. Example of the `link_metro` opcode.

```
<CsoundSynthesizer>
<CsLicense>
Run the Ableton Link "LinkHut" example application, or some other
Ableton Link peer, while you run this CSD to see what happens.
</CsLicense>
<CsOptions>
-m0 -d -odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 10
```

```
nchnls = 2

alwayson "LinkMonitor"

gi_peer link_create 72
gk_beat init 0

instr Beep
asignal oscils 20000, p4, 0
outs asignal, asignal
endin

instr TempoChange
link_tempo_set gi_peer, 80
endin

instr LinkEnable
i_enabled = p4
link_enable gi_peer, i_enabled
endin

instr LinkMonitor
i_kperiod_seconds = ksmpls / sr
printf_i "kperiod: %9.6f seconds.\n", 1, i_kperiod_seconds
printf_i "gi_peer: %g\n", 1, gi_peer
link_enable gi_peer, 1
k_trigger, gk_beat, k_phase, k_time link_metro gi_peer, 1
k_peers link_peers gi_peer
k_tempo link_tempo_get gi_peer
k_enabled link_is_enabled gi_peer
k_hz = 1000
if floor(gk_beat % 4) == 0 then
k_hz = 3000
else
k_hz = 2000
endif
schedkwhen k_trigger, 0, 1, "Beep", 0, 0.01, k_hz
printf "LinkMonitor: gi_peer: %g k_enabled: %9.4f k_trigger: %9.4f beat: %9.4f k_phase: %9.4f time: %9.4f\n", 1, gi_peer, k_enabled, k_trigger, gk_beat, k_phase, k_time
endin
</CsInstruments>
<CsScore>
f 0 360
i "TempoChange" 10 80
i "LinkEnable" 20 1 0
i "LinkEnable" 30 1 1
</CsScore>
</CsoundSynthesizer>
```

## See Also

See the section *Ableton Link Opcodes* for more information.

## Credits

By: Michael Gogins 2017

# link\_peers

link\_peers — Returns the number of peers in the session.

## Description

Returns the number of peers in the global network Ableton Link session.

## Syntax

```
k_count link_peers i_peer
```

## Initialization

*i\_peer* -- The Ableton Link peer object.

## Performance

*k\_count* -- The current number of peers in the global network Ableton Link session.

## Examples

Here is an example of the link\_peers opcode. It uses the file *ableton\_link.csd* [examples/ableton\_link.csd].

### Example 465. Example of the link\_peers opcode.

```
<CsoundSynthesizer>
<CsLicense>
Run the Ableton Link "LinkHut" example application, or some other
Ableton Link peer, while you run this CSD to see what happens.
</CsLicense>
<CsOptions>
-m0 -d -odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 10
nchnls = 2

alwayson "LinkMonitor"

gi_peer link_create 72
gk_beat init 0

instr Beep
asignal oscils 20000, p4, 0
outs asignal, asignal
endin

instr TempoChange
link_tempo_set gi_peer, 80
endin

instr LinkEnable
i_enabled = p4
```

```
link_enable gi_peer, i_enabled
endin

instr LinkMonitor
i_kperiod_seconds = ksmps / sr
printf_i "kperiod: %9.6f seconds.\n", 1, i_kperiod_seconds
printf_i "gi_peer: %g\n", 1, gi_peer
link_enable gi_peer, 1
k_trigger, gk_beat, k_phase, k_time link_metro gi_peer, 1
k_peers link_peers gi_peer
k_tempo link_tempo_get gi_peer
k_enabled link_is_enabled gi_peer
k_hz = 1000
if floor(gk_beat % 4) == 0 then
k_hz = 3000
else
k_hz = 2000
endif
schedkwhen k_trigger, 0, 1, "Beep", 0, 0.01, k_hz
printf "LinkMonitor: gi_peer: %g k_enabled: %9.4f k_trigger: %9.4f beat: %9.4f k_phase: %9.4f time: %9.4f\n", 1, k_enabled, k_trigger, gk_beat, k_phase, k_time
endin
</CsInstruments>
<CsScore>
f 0 360
i "TempoChange" 10 80
i "LinkEnable" 20 1 0
i "LinkEnable" 30 1 1
</CsScore>
</CsoundSynthesizer>
```

## See Also

See the section *Ableton Link Opcodes* for more information.

## Credits

By: Michael Gogins 2017

# link\_tempo\_get

link\_tempo\_get — Returns the current tempo of the global network Ableton Link session.

## Description

Returns the current tempo of the global network Ableton Link session.

## Syntax

```
k_bpm link_tempo_get i_peer
```

## Initialization

*i\_peer* -- The Ableton Link peer object.

## Performance

*k\_bpm* --The current tempo of the global network Ableton Link session.

## Examples

Here is an example of the link\_tempo\_get opcode. It uses the file *ableton\_link.csd* [examples/ableton\_link.csd].

### Example 466. Example of the link\_tempo\_get opcode.

```
<CsoundSynthesizer>
<CsLicense>
Run the Ableton Link "LinkHut" example application, or some other
Ableton Link peer, while you run this CSD to see what happens.
</CsLicense>
<CsOptions>
-m0 -d -odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 10
nchnls = 2

alwayson "LinkMonitor"

gi_peer link_create 72
gk_beat init 0

instr Beep
asignal oscils 20000, p4, 0
outs asignal, asignal
endin

instr TempoChange
link_tempo_set gi_peer, 80
endin
```

```
instr LinkEnable
i_enabled = p4
link_enable gi_peer, i_enabled
endin

instr LinkMonitor
i_kperiod_seconds = ksmps / sr
printf_i "kperiod: %9.6f seconds.\n", 1, i_kperiod_seconds
printf_i "gi_peer: %g\n", 1, gi_peer
link_enable gi_peer, 1
k_trigger, gk_beat, k_phase, k_time link_metro gi_peer, 1
k_peers link_peers gi_peer
k_tempo link_tempo_get gi_peer
k_enabled link_is_enabled gi_peer
k_hz = 1000
if floor(gk_beat % 4) == 0 then
k_hz = 3000
else
k_hz = 2000
endif
schedkwhen k_trigger, 0, 1, "Beep", 0, 0.01, k_hz
printf "LinkMonitor: gi_peer: %g k_enabled: %9.4f k_trigger: %9.4f beat: %9.4f k_phase: %9.4f time: %9.4f\n", 1, k_enabled, k_trigger, gk_beat, k_phase, k_time, k_hz
endin
</CsInstruments>
<CsScore>
f 0 360
i "TempoChange" 10 80
i "LinkEnable" 20 1 0
i "LinkEnable" 30 1 1
</CsScore>
</CsoundSynthesizer>
```

## See Also

See the section *Ableton Link Opcodes* for more information.

## Credits

By: Michael Gogins 2017

# link\_tempo\_set

link\_tempo\_set — Sets the tempo.

## Description

Sets the local tempo if this peer is not enabled; sets the tempo of the global network Ableton Link session if this peer is enabled.

## Syntax

```
link_tempo_set i_peer, k_bpm [, k_at_time_seconds]
```

## Initialization

*i\_peer* -- The Ableton Link peer object.

*k\_bpm* -- The requested tempo in beats per minute.

*k\_at\_time\_seconds* -- The session time at which the indicated tempo will take effect. The default value is the current session time.

## Examples

Here is an example of the link\_tempo\_set opcode. It uses the file *ableton\_link.csd* [examples/ableton\_link.csd].

### Example 467. Example of the link\_tempo\_set opcode.

```
<CsoundSynthesizer>
<CsLicense>
Run the Ableton Link "LinkHut" example application, or some other
Ableton Link peer, while you run this CSD to see what happens.
</CsLicense>
<CsOptions>
-m0 -d -odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 10
nchnls = 2

alwayson "LinkMonitor"

gi_peer link_create 72
gk_beat init 0

instr Beep
asignal oscils 20000, p4, 0
outs asignal, asignal
endin

instr TempoChange
link_tempo_set gi_peer, 80
endin
```

```
instr LinkEnable
i_enabled = p4
link_enable gi_peer, i_enabled
endin

instr LinkMonitor
i_kperiod_seconds = ksmps / sr
printf_i "kperiod: %9.6f seconds.\n", 1, i_kperiod_seconds
printf_i "gi_peer: %g\n", 1, gi_peer
link_enable gi_peer, 1
k_trigger, gk_beat, k_phase, k_time link_metro gi_peer, 1
k_peers link_peers gi_peer
k_tempo link_tempo_get gi_peer
k_enabled link_is_enabled gi_peer
k_hz = 1000
if floor(gk_beat % 4) == 0 then
k_hz = 3000
else
k_hz = 2000
endif
schedkwhen k_trigger, 0, 1, "Beep", 0, 0.01, k_hz
printf "LinkMonitor: gi_peer: %g k_enabled: %9.4f k_trigger: %9.4f beat: %9.4f k_phase: %9.4f time: %9.4f\n", 1, k_enabled, k_trigger, gk_beat, k_phase, k_time
endin
</CsInstruments>
<CsScore>
f 0 360
i "TempoChange" 10 80
i "LinkEnable" 20 1 0
i "LinkEnable" 30 1 1
</CsScore>
</CsoundSynthesizer>
```

## See Also

See the section *Ableton Link Opcodes* for more information.

## Credits

By: Michael Gogins 2017

# linrand

linrand — Linear distribution random number generator (positive values only).

## Description

Linear distribution random number generator (positive values only). This is an x-class noise generator.

## Syntax

```
ares linrand krange
ires linrand krange
kres linrand krange
```

## Performance

*krange* -- the range of the random numbers (0 - *krange*). Outputs only positive numbers.

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the linrand opcode. It uses the file *linrand.csd* [examples/linrand.csd].

### Example 468. Example of the linrand opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o linrand.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; every run time same values

klin linrand 100
printk .2, klin ; look
```

```

aout oscili 0.8, 440+klin, 1 ; & listen
outs aout, aout
endin

instr 2 ; every run time different values

seed 0
klin linrand 100
printk .2, klin ; lock
aout oscili 0.8, 440+klin, 1 ; & listen
outs aout, aout
endin

</CsInstruments>
<CsScore>
; sine wave
f 1 0 16384 10 1

i 1 0 2
i 2 3 2
e
</CsScore>
</CsoundSynthesizer>

```

Its output should include a line like this:

```

i 1 time 0.00033: 13.54770
i 1 time 0.20033: 32.38746
i 1 time 0.40033: 47.69304
i 1 time 0.60033: 19.82218
i 1 time 0.80033: 42.98293
i 1 time 1.00000: 81.13174
i 1 time 1.20033: 47.39585
i 1 time 1.40033: 12.53248
i 1 time 1.60033: 35.70722
i 1 time 1.80000: 65.25774
i 1 time 2.00000: 23.24811
Seeding from current time 392575384
i 2 time 3.00033: 23.05609
i 2 time 3.20033: 76.15114
i 2 time 3.40033: 22.78861
i 2 time 3.60000: 0.79064
i 2 time 3.80033: 43.49438
i 2 time 4.00000: 34.10963
i 2 time 4.20000: 31.88702
i 2 time 4.40033: 59.78054
i 2 time 4.60033: 4.96821
i 2 time 4.80033: 24.69674
i 2 time 5.00000: 21.88815

```

## See Also

*seed*, *betarand*, *bexprnd*, *cauchy*, *exprand*, *gauss*, *pcauchy*, *poisson*, *trirand*, *unirand*, *weibull*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

# linseg

linseg — Trace a series of line segments between specified points.

## Description

Trace a series of line segments between specified points.

## Syntax

```
ares linseg ia, idur1, ib [, idur2] [, ic] [...]  
kres linseg ia, idur1, ib [, idur2] [, ic] [...]
```

## Initialization

*ia* -- starting value.

*ib*, *ic*, etc. -- value after *dur1* seconds, etc.

*idur1* -- duration in seconds of first segment. A zero or negative value will cause all initialization to be skipped.

*idur2*, *idur3*, etc. -- duration in seconds of subsequent segments. A zero or negative value will terminate the initialization process with the preceding point, permitting the last-defined line or curve to be continued indefinitely in performance. The default is zero.

## Performance

These units generate control or audio signals whose values can pass through 2 or more specified points. The sum of *dur* values may or may not equal the instrument's performance time: a shorter performance will truncate the specified pattern, while a longer one will cause the last value to be repeated until the end of the note.

## Examples

Here is an example of the linseg opcode. It uses the file *linseg.csd* [examples/linseg.csd].

### Example 469. Example of the linseg opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
-odac      ;;realtime audio out  
;-iadc     ;;uncomment -iadc if realtime audio input is needed too  
; For Non-realtime ouput leave only the line below:  
;-o linseg.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>
```

```

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine ftgen 0, 0, 2^10, 10, 1

instr 1

kcps = cpspch(p4)
kenv linseg 0, 0.25, 1, 0.75, 0 ; together = 1 sec
asig poscil kenv, kcps, giSine
outs asig, asig

endin

instr 2 ; scaling to duration

kcps = cpspch(p4)
kenv linseg 0, p3*0.25, 1, p3*0.75, 0
asig poscil kenv, kcps, giSine
outs asig, asig

endin

instr 3 ; with negative value

kcps = cpspch(p4)
aenv linseg 0, 0.1, 1, 0.5, -0.9, 0.4, 0
asig poscil aenv, kcps, giSine
outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 1 7.00 ; = 1 sec, p3 fits exactly
i 1 2 2 7.00 ; = 2 sec, p3 truncated at 1 sec

i 2 4 1 7.00 ; scales to duration
i 2 6 2 7.00 ; of p3

i 3 9 2 7.00
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*expon, expseg, expsegb, expsegr, cosseg, line, linsegr transeg transegb*

## Credits

Author: Barry L. Vercoe

# linsegb

linsegb — Trace a series of line segments between specified absolute points.

## Description

Trace a series of line segments between specified absolute points.

## Syntax

```
ares linsegb ia, itim1, ib [, itim2] [, ic] [...]  
kres linsegb ia, itim1, ib [, itim2] [, ic] [...]
```

## Initialization

*ia* -- starting value.

*ib*, *ic*, etc. -- value at *tim1* seconds, etc.

*itim1* -- time in seconds of end of first segment. A zero or negative value will cause all initialization to be skipped.

*itim2*, *itim3*, etc. -- time in seconds at the end of subsequent segments.

## Performance

These units generate control or audio signals whose values can pass through 2 or more specified points. The last *tim* value may or may not equal the instrument's performance time: a shorter performance will truncate the specified pattern, while a longer one will cause the last value to be repeated until the end of the note.

## Examples

Here is an example of the linsegb opcode. It uses the file *linsegb.csd* [examples/linsegb.csd].

### Example 470. Example of the linsegb opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
-odac      ;;realtime audio out  
;-iadc     ;;uncomment -iadc if realtime audio input is needed too  
; For Non-realtime ouput leave only the line below:  
;-o linseg.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>  
  
sr = 44100  
ksmps = 32  
nchnls = 2
```

```
Odbfs = 1

giSine ftgen 0, 0, 2^10, 10, 1

instr 1

kcps = cpspch(p4)
kenv linsegb 0, 0.25, 1, 1, 0
asig poscil kenv, kcps, giSine
outs asig, asig

endin

instr 2 ; scaling to duration

kcps = cpspch(p4)
kenv linseg 0, p3*0.25, 1, p3, 0
asig poscil kenv, kcps, giSine
outs asig, asig

endin

</CsInstruments>
<CsScore>

i 1 0 1 7.00 ; = 1 sec, p3 fits exactly
i 1 2 2 7.00 ; = 2 sec, p3 truncated at 1 sec

i 2 4 1 7.00 ; scales to duration
i 2 6 2 7.00 ; of p3

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*expon, expseg, expsegr, line, linseg linsegr transeg*

## Credits

Author: Victor Lazzarini  
June 2011

New in version 5.14

# linsegr

linsegr — Trace a series of line segments between specified points including a release segment.

## Description

Trace a series of line segments between specified points including a release segment.

## Syntax

```
ares linsegr ia, idur1, ib [, idur2] [, ic] [...], irel, iz  
kres linsegr ia, idur1, ib [, idur2] [, ic] [...], irel, iz
```

## Initialization

*ia* -- starting value.

*ib*, *ic*, etc. -- value after *dur1* seconds, etc.

*idur1* -- duration in seconds of first segment. A zero or negative value will cause all initialization to be skipped.

*idur2*, *idur3*, etc. -- duration in seconds of subsequent segments. A zero or negative value will terminate the initialization process with the preceding point, permitting the last-defined line or curve to be continued indefinitely in performance. The default is zero.

*irel*, *iz* -- duration in seconds and final value of a note releasing segment.

For Csound versions prior to 5.00, the release time cannot be longer than  $32767/kr$  seconds. This limit has been extended to  $(2^{31}-1)/kr$ .

## Performance

These units generate control or audio signals whose values can pass through 2 or more specified points. The sum of *dur* values may or may not equal the instrument's performance time: a shorter performance will truncate the specified pattern, while a longer one will cause the last-defined segment to continue on in the same direction.

*linsegr* is amongst the Csound “r” units that contain a note-off sensor and release time extender. When each senses an event termination or MIDI noteoff, it immediately extends the performance time of the current instrument by *irel* seconds, and sets out to reach the value *iz* by the end of that period (no matter which segment the unit is in). “r” units can also be modified by MIDI noteoff velocities. For two or more extenders in an instrument, extension is by the greatest period.

You can use other pre-made envelopes which start a release segment upon receiving a note off message, like *linenr* and *expsegr*, or you can construct more complex envelopes using *xtratim* and *release*. Note that you don't need to use *xtratim* if you are using *linsegr*, since the time is extended automatically.

## Examples

Here is an example of the *linsegr* opcode. It uses the file *linsegr.csd* [examples/linsegr.csd].

### Example 471. Example of the `linsegr` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0   ;;realtime audio out and realtime midi in
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o linsegr.wav -W   ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

icps cpsmidi
iamp ampmidi .3

kenv linsegr 1, .05, 0.5, 1, 0
asig pluck kenv, icps, 200, 1, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 4096 10 1 ;sine wave

f0 30 ;runs 30 seconds
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*linenr*, *expsegr*, *envlpxr*, *mxadsr*, *madsr* *expon*, *expseg*, *expsega* *line*, *linseg*, *xtratim*, *transegr*

## Credits

Author: Barry L. Vercoe

December 2002, December 2006. Thanks to Istvan Varga, added documentation about the maximum release time.

New in Csound 3.47

# liveconv

liveconv — Partitioned convolution with dynamically reloadable impulse response

## Description

Computationally efficient, partitioned convolution, using a function table as impulse response (IR) source, similar to the *ftconv* opcode. The *liveconv* opcode allows dynamic reload of IR data at any time while the convolution is running, controlled by the *kupdate* parameter. Due to the manner in which the IR is updated, the operation can be done without audio artifacts in the convolution output.

The IR table is read partition by partition after signalling an update. The first *ksmps* samples must be available when the update flag is set. The rest of the table can be continuously filled in consecutive passes. There may be a new update for every partition, allowing as many as *ftlen/iplen* simultaneous updates (*ftlen* is the length of the IR table).

The low-latency, dynamic behavior of *liveconv* makes it ideal for convolution with live sampled impulse responses, and/or real-time impulse response transformations and modifications.

## Syntax

```
ares liveconv ain, ift, iplen, kupdate, kclear
```

## Initialization

*ift* -- table number for storing the impulse response (IR) for convolution. The table may be filled with new data at any time while the convolution is running.

*iplen* -- length of impulse response partition in samples; must be an integer power of two. Lower settings allow for shorter output delay, but will increase CPU usage.

## Performance

*ain* -- input signal.

*ares* -- output signal.

*kupdate* -- flag indicating whether the IR table should be updated. If *kupdate*=1 the IR table *ift* is loaded partition by partition, starting with the next partition. If *kupdate*=-1 the IR table *ift* is unloaded (cleared to zero) partition by partition, starting with the next partition. Other values have no effect.

*kclear* -- flag for clearing all internal buffers. If *kclear* has any value != zero, the internal buffers are cleared immediately. This operation is not free of artifacts.

## Example

Here is an example of the *liveconv* opcode. It uses the file *liveconv.csd* [examples/liveconv.csd].

### Example 472. Example of the liveconv opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
-odac ;realtime audio out
</CsOptions>
<CsInstruments>

  sr = 44100
  nchnls = 2
  odbfs = 1

  ; empty IR table
  giIR_record ftgen 0, 0, 131072, 2, 0

  ; Record impulse response
  instr 13

  p3 = ftden(giIR_record)/sr
  iskip = p4
  irlen = p5
  a1 diskin2 "fox.wav", 1, iskip

  ; Fill IR table with segment from audio file
  amp linseg 0, 0.1, 1, irlen, 1, 0.1, 0, 1, 0
  andx_IR line 0, 1, 1/(ftlen(giIR_record)/sr)
  tablew a1*amp, andx_IR, giIR_record, 1
  outch 1, a1*amp ; output the IR
  ktrig init 1
  if ktrig > -1 then
    chnset ktrig, "conv_update"
    ktrig -= 1
  endif

  endin

  ; The convolver
  instr 14

  ain diskin2 "beats.wav", 1, 0, 1
  kupdate chnget "conv_update"
  aconv liveconv ain, giIR_record, 2048, kupdate, 0
  outch 2, aconv*0.009 ; output the convolution response
  endin

</CsInstruments>
<CsScore>
; record impulse response
; skip IR_dur
i13 0 1 0.0 0.5
i13 2 1 0.5 0.5
i13 4 1 1.0 0.5
i13 6 1 1.5 0.5
i13 8 1 2.0 0.75
i13 10 1 2.38 0.25

; convolve
i14 0.0 11.65

e

</CsScore>
</CsoundSynthesizer>

```

## See also

*ficonv tvconv pconvolve, convolve,*

## Credits

Author: Sigurd Saue, Oeyvind Brandtsegg  
2017

# locsend

locsend — Distributes the audio signals of a previous *locsig* opcode.

## Description

*locsend* depends upon the existence of a previously defined *locsig*. The number of output signals must match the number in the previous *locsig*. The output signals from *locsend* are derived from the values given for distance and reverb in the *locsig* and are ready to be sent to local or global reverb units (see example below). The reverb amount and the balance between the 2 or 4 channels are calculated in the same way as described in the Dodge book (an essential text!).

## Syntax

```
a1, a2 locsend
```

```
a1, a2, a3, a4 locsend
```

## Examples

```

asig some audio signal
kdegree      line  0, p3, 360
kdistance    line  1, p3, 10
a1, a2, a3, a4  locsig asig, kdegree, kdistance, .1
ar1, ar2, ar3, ar4 locsend
ga1 = ga1+ar1
ga2 = ga2+ar2
ga3 = ga3+ar3
ga4 = ga4+ar4
                                outq  a1, a2, a3, a4
endin

instr 99 ; reverb instrument
a1      reverb2 ga1, 2.5, .5
a2      reverb2 ga2, 2.5, .5
a3      reverb2 ga3, 2.5, .5
a4      reverb2 ga4, 2.5, .5
                                outq  a1, a2, a3, a4

ga1=0
ga2=0
ga3=0
ga4=0

```

In the above example, the signal, *asig*, is sent around a complete circle once during the duration of a note while at the same time it becomes more and more “distant” from the listeners' location. *locsig* sends the appropriate amount of the signal internally to *locsend*. The outputs of the *locsend* are added to global accumulators in a common Csound style and the global signals are used as inputs to the reverb units in a separate instrument. For an example, see *locsig*.

*locsig* is useful for quad and stereo panning as well as fixed placed of sounds anywhere between two loudspeakers. Below is an example of the fixed placement of sounds in a stereo field. It uses the file *locsend\_stereo.csd* [examples/locsend\_stereo.csd].

### Example 473. Example of the locsend opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o locsend_stereo.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

ga1 init 0
ga2 init 0

instr 1

krevsend = p4
aout diskin2 "beats.wav", 1, 0, 1
kdegree line 0, p3, 180 ;left to right
kdistance line 1, p3, 30
a1, a2 locsig aout, kdegree, kdistance, p4
ar1, ar2 locsend
ga1 = ga1+ar1
ga2 = ga2+ar2
      outs a1, a2

endin

instr 99 ; reverb instrument
a1 reverb2 ga1, 2.5, .5
a2 reverb2 ga2, 2.5, .5
      outs a1, a2
ga1 = 0
ga2 = 0

endin
</CsInstruments>
<CsScore>
; sine wave.
f 1 0 16384 10 1

i 1 0 4 .1 ;with reverb
i 1 + 4 0 ;no reverb
i99 0 7
e
</CsScore>
</CsoundSynthesizer>

```

A few notes:

```

;place the sound in the left speaker and near:
i1 0 1 0 1

```

```

;place the sound in the right speaker and far:
i1 1 1 90 25

```

```

;place the sound equally between left and right and in the middle ground distance:
i1 2 1 45 12
e

```

The next example shows a simple intuitive use of the distance value to simulate Doppler shift. The same value is used to scale the frequency as is used as the distance input to *locsig*.

```
kdistance      line  1, p3, 10
kfreq = (ifreq * 340) / (340 + kdistance)
asig          oscili iamp, kfreq, 1
kdegree       line  0, p3, 360
a1, a2, a3, a4  locsig asig, kdegree, kdistance, .1
ar1, ar2, ar3, ar4 locsend
```

## See Also

*locsig*

## Credits

Author: Richard Karpen  
Seattle, WA USA  
1998

New in Csound version 3.48

# locsig

locsig — Takes an input signal and distributes between 2 or 4 channels.

## Description

*locsig* takes an input signal and distributes it among 2 or 4 channels using values in degrees to calculate the balance between adjacent channels. It also takes arguments for distance (used to attenuate signals that are to sound as if they are some distance further than the loudspeaker itself), and for the amount the signal that will be sent to reverberators. This unit is based upon the example in the Charles Dodge/Thomas Jerse book, *Computer Music*, page 320.

## Syntax

```
a1, a2 locsig asig, kdegree, kdistance, kreverbsend
```

```
a1, a2, a3, a4 locsig asig, kdegree, kdistance, kreverbsend
```

## Performance

*kdegree* -- value between 0 and 360 for placement of the signal in a 2 or 4 channel space configured as: a1=0, a2=90, a3=180, a4=270 (kdegree=45 would balanced the signal equally between a1 and a2). *locsig* maps *kdegree* to sin and cos functions to derive the signal balances (e.g.: asig=1, kdegree=45, a1=a2=.707).

*kdistance* -- value  $\geq 1$  used to attenuate the signal and to calculate reverb level to simulate distance cues. As *kdistance* gets larger the sound should get softer and somewhat more reverberant (assuming the use of *locsend* in this case).

*kreverbsend* -- the percentage of the direct signal that will be factored along with the distance and degree values to derive signal amounts that can be sent to a reverb unit such as *reverb*, or *reverb2*.

## Examples

Here is an example of the locsig opcode. It uses the file *locsig\_quad.csd* [examples/locsig\_quad.csd].

### Example 474. Example of the locsig opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o locsig_quad.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 4
0dbfs = 1
```

```

ga1 init 0
ga2 init 0
ga3 init 0
ga4 init 0

instr 1

krevsend = p4
aout diskin2 "beats.wav", 1, 0, 1
kdegree line 0, p3, 360 ;full circle
kdistance line 1, p3, 1
a1, a2, a3, a4 locsig aout, kdegree, kdistance, krevsend
ar1, ar2, ar3, ar4 locsend

ga1 = ga1+ar1
ga2 = ga2+ar2
ga3 = ga3+ar3
ga4 = ga4+ar4
outq a1, a2, a3, a4

endin

instr 99 ; reverb instrument
a1 reverb2 ga1, 3.5, .5
a2 reverb2 ga2, 3.5, .5
a3 reverb2 ga3, 3.5, .5
a4 reverb2 ga4, 3.5, .5
outq a1, a2, a3, a4

ga1 = 0
ga2 = 0
ga3 = 0
ga4 = 0

endin
</CsInstruments>
<CsScore>
; sine wave.
f 1 0 16384 10 1

i 1 0 14 .1 ;with reverb
i 1 14 14 0 ;no reverb
i 99 0 36
e
</CsScore>
</CsSoundSynthesizer>

```

In the above example, the signal, *aout*, is sent around a complete circle once during the duration of a note while at the same time it becomes more and more "distant" from the listeners' location. *locsig* sends the appropriate amount of the signal internally to *locsend*. The outputs of the *locsend* are added to global accumulators in a common Csound style and the global signals are used as inputs to the reverb units in a separate instrument.

*locsig* is useful for quad and stereo panning as well as fixed placement of sounds anywhere between two loudspeakers. Below is an example of the fixed placement of sounds in a stereo field.

```

instr 1
  a1, a2          locsig asig, p4, p5, .1
  ar1, ar2       locsend
  ga1 = ga1+ar1
  ga2 = ga2+ar2
  outs a1, a
endin
instr 99

```

```
    ; reverb...
  endin
```

A few notes:

```
    ;place the sound in the left speaker and near:
  i1 0 1 0 1
```

```
    ;place the sound in the right speaker and far:
  i1 1 1 90 25
```

```
    ;place the sound equally between left and right and in the middle ground distance:
  i1 2 1 45 12
  e
```

The next example shows a simple intuitive use of the distance value to simulate Doppler shift. The same value is used to scale the frequency as is used as the distance input to *locsig*.

```
  kdistance      line 1, p3, 10
  kfreq = (ifreq * 340) / (340 + kdistance)
  asig          oscili iamp, kfreq, 1
  kdegree      line 0, p3, 360
  a1, a2, a3, a4  locsig asig, kdegree, kdistance, .1
  ar1, ar2, ar3, ar4 locsend
```

## See Also

*locsend*

## Credits

Author: Richard Karpen  
Seattle, WA USA  
1998

New in Csound version 3.48

# log

log — Returns a natural log of a number, or an array (with optional arbitrary base).

## Description

Returns the natural log of  $x$  ( $x$  positive only). In the case of an array input, the operation can have an optional arbitrary base.

The argument value is restricted for *log*, *log10*, and *sqrt*.

## Syntax

```
log(x) (no rate
      restriction)

log(k/i[]) (k- or i-arrays )

kout[]log kin[],ibas
```

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression. In the case of an array input, *ibas* is the optional arbitrary base, which defaults to *e* (natural log base).

## Examples

Here is an example of the log opcode. It uses the file *log.csd* [examples/log.csd].

### Example 475. Example of the log opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o log.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  i1 = log(8)
  print i1
endin

</CsInstruments>
```

```
<CsScore>
; Play Instrument #1 for one second.
i 1 0 1
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include a line like this:

```
instr 1: i1 = 2.079
```

## See Also

*abs, exp, frac, int, log10, i, sqrt*

## Credits

Written by John ffitch.

New in version 3.47

Example written by Kevin Conder.

# log10

log10 — Returns a base 10 log.

## Description

Returns the base 10 log of  $x$  ( $x$  positive only).

The argument value is restricted for *log*, *log10*, and *sqrt*.

## Syntax

`log10(x)` (no rate restriction)

`log10(k/i[])` (k- or i-arrays )

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

## Examples

Here is an example of the log10 opcode. It uses the file *log10.csd* [examples/log10.csd].

### Example 476. Example of the log10 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o log10.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  i1 = log10(8)
  print i1
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
e
```

```
</CsScore>  
</CsoundSynthesizer>
```

Its output should include a line like this:

```
instr 1: i1 = 0.903
```

## See Also

*abs, exp, frac, int, log, log, i, sqrt*

## Credits

Written by John fitch.

New in version 3.47

Example written by Kevin Conder.

# log2

log2 — Returns a base 2 log.

## Description

Returns the base 2 log of  $x$  ( $x$  positive only).

The argument value is restricted for *log*, *log2*, and *sqrt*.

## Syntax

```
log2(x) (no rate  
restriction)  
  
log2(k/i[]) (k- or i-arrays )
```

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

## Examples

Here is an example of the log2 opcode. It uses the file *log2.csd* [examples/log2.csd].

### Example 477. Example of the log2 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsoundOptions>  
; Select audio/midi flags here according to platform  
; Audio out  Audio in  
-odac          -iadc      ;;RT audio I/O  
; For Non-realtime output leave only the line below:  
; -o log10.wav -W ;; for file output any platform  
</CsoundOptions>  
<CsoundInstruments>  
  
; Initialize the global variables.  
sr = 44100  
kr = 4410  
ksmps = 10  
nchnls = 1  
  
; Instrument #1.  
instr 1  
  i1 = log2(8)  
  print i1  
endin  
  
</CsoundInstruments>  
<CsoundScore>  
  
; Play Instrument #1 for one second.  
i 1 0 1
```

e

```
</CsScore>  
</CsoundSynthesizer>
```

Its output should include a line like this:

```
instr 1: i1 = 3.000
```

## See Also

*abs, exp, frac, int, log, log, i, sqrt*

## Credits

Written by John ffitth.

New in version 5.17.12

# logbtwo

logbtwo — Performs a logarithmic base two calculation.

## Description

Performs a logarithmic base two calculation.

## Syntax

```
logbtwo(x) (init-rate or control-rate args only)
```

## Performance

*logbtwo()* returns the logarithm base two of *x*. The range of values admitted as argument is .25 to 4 (i.e. from -2 octave to +2 octave response). This function is the inverse of *powoftwo()*.

These functions are fast, because they read values stored in tables. Also they are very useful when working with tuning ratios. They work at i- and k-rate.

## Examples

Here is an example of the logbtwo opcode. It uses the file *logbtwo.csd* [examples/logbtwo.csd].

### Example 478. Example of the logbtwo opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o logbtwo.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  i1 = logbtwo(3)
  print i1
endin

</CsInstruments>
<CsScore>
```

```
; Play Instrument #1 for one second.  
i 1 0 1  
e
```

```
</CsScore>  
</CsoundSynthesizer>
```

Its output should include a line like this:

```
instr 1: i1 = 1.585
```

## See Also

*powoftwo*

## Credits

Author: Gabriel Maldonado  
Italy  
June 1998

Author: John ffitch  
University of Bath, Codemist, Ltd.  
Bath, UK  
July 1999

Example written by Kevin Conder.

New in Csound version 3.57

# logcurve

logcurve — This opcode implements a formula for generating a normalised logarithmic curve in range 0 - 1. It is based on the Max / MSP work of Eric Singer (c) 1994.

## Description

Generates a logarithmic curve in range 0 to 1 of arbitrary steepness. Steepness index equal to or lower than 1.0 will result in Not-a-Number errors and cause unstable behavior.

The formula used to calculate the curve is:

$$\log(x * (y-1)+1) / (\log(y))$$

where x is equal to *kindex* and y is equal to *ksteepness*.

## Syntax

```
kout logcurve kindex, ksteepness
```

## Performance

*kindex* -- Index value. Expected range 0 to 1.

*ksteepness* -- Steepness of the generated curve. Values closer to 1.0 result in a straighter line while larger values steepen the curve.

*kout* -- Scaled output.

## Examples

Here is an example of the logcurve opcode. It uses the file *logcurve.csd* [examples/logcurve.csd].

### Example 479. Example of the logcurve opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  Silent
-odac      -iadc      -d      ;;;realtime output
</CsOptions>
<CsInstruments>

sr = 48000
ksmps = 100
nchnls = 2

instr 1 ; logcurve test

kmod phasor 1/p3

kout logcurve kmod, p4
```

```
printks "kmod = %f kout = %f\\n", 0.1, kmod, kout

endin

</CsInstruments>
<CsScore>

i1 0 10 2
i1 10 10 30
i1 20 10 0.5

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*scale, gainslider, expcurve*

## Credits

Author: David Akbari  
October  
2006

# loop\_ge

`loop_ge` — Looping constructions.

## Description

Construction of looping operations.

## Syntax

```
loop_ge indx, idecr, imin, label
```

```
loop_ge kndx, kdecr, kmin, label
```

## Initialization

*indx* -- i-rate variable to count loop.

*idecr* -- value to decrement the loop.

*imin* -- minimum value of loop index.

## Performance

*kndx* -- k-rate variable to count loop.

*kdecr* -- value to decrement the loop.

*kmin* -- minimum value of loop index.

The actions of **loop\_ge** are equivalent to the code

```
indx = indx - idecr
if (indx >= imin) igoto label
```

or

```
kndx = kndx - kdecr
if (kndx >= kmin) kgoto label
```

## See Also

*loop\_gt*, *loop\_le* and *loop\_lt*.

More information on this opcode: [http://www.csoundjournal.com/2006summer/controlFlow\\_part2.html](http://www.csoundjournal.com/2006summer/controlFlow_part2.html), written by Steven Yi. And in the Floss Manuals: [http://en.flossmanuals.net/csound/ch018\\_c-control-structures](http://en.flossmanuals.net/csound/ch018_c-control-structures) [[http://en.flossmanuals.net/csound/ch018\\_c-control-structures](http://en.flossmanuals.net/csound/ch018_c-control-structures)]

## Credits

Istvan Varga. 2006

New in Csound version 5.01

# loop\_gt

loop\_gt — Looping constructions.

## Description

Construction of looping operations.

## Syntax

```
loop_gt  indx, idecr, imin, label
```

```
loop_gt  kndx, kdecr, kmin, label
```

## Initialization

*indx* -- i-rate variable to count loop.

*idecr* -- value to decrement the loop.

*imin* -- minimum value of loop index.

## Performance

*kndx* -- k-rate variable to count loop.

*kdecr* -- value to decrement the loop.

*kmin* -- minimum value of loop index.

The actions of **loop\_gt** are equivalent to the code

```
indx = indx - idecr
if (indx > imin) igoto label
```

or

```
kndx = kndx - kdecr
if (kndx > kmin) kgoto label
```

## See Also

*loop\_ge*, *loop\_le* and *loop\_lt*.

More information on this opcode: [http://www.csoundjournal.com/2006summer/controlFlow\\_part2.html](http://www.csoundjournal.com/2006summer/controlFlow_part2.html), written by Steven Yi. And in the Floss Manuals: [http://en.flossmanuals.net/csound/ch018\\_c-control-structures](http://en.flossmanuals.net/csound/ch018_c-control-structures) [[http://en.flossmanuals.net/csound/ch018\\_c-control-structures](http://en.flossmanuals.net/csound/ch018_c-control-structures)]

## Credits

Istvan Varga.

New in Csound version 5.01

# loop\_le

loop\_le — Looping constructions.

## Description

Construction of looping operations.

## Syntax

```
loop_le indx, incr, imax, label
```

```
loop_le kndx, kncr, kmax, label
```

## Initialization

*indx* -- i-rate variable to count loop.

*incr* -- value to increment the loop.

*imax* -- maximum value of loop index.

## Performance

*kndx* -- k-rate variable to count loop.

*knrc* -- value to increment the loop.

*kmax* -- maximum value of loop index.

The actions of **loop\_le** are equivalent to the code

```
indx = indx + incr  
if (indx <= imax) igoto label
```

or

```
kndx = kndx + knrc  
if (kndx <= kmax) kgoto label
```

## Examples

Here is an example of the loop\_le opcode. It uses the file *loop\_le.csd* [examples/loop\_le.csd].

### Example 480. Example of the loop\_le opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>
```

```

; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o loop_le.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

seed 0
gisine ftgen 0, 0, 2^10, 10, 1

instr 1 ;master instrument

ininstr = 5 ;number of called instances
indx = 0
loop:
    prints "play instance %d\\n", indx
    ipan random 0, 1
    ifreq random 100, 1000
    iamp = 1/ininstr
    event_i "i", 10, 0, p3, iamp, ifreq, ipan
    loop_le indx, 1, ininstr, loop

endin

instr 10

ipeak random 0, 1 ;where is the envelope peak
asig poscil3 p4, p5, gisine
aenv transeg 0, p3*ipeak, 6, 1, p3-p3*ipeak, -6, 0
aL,aR pan2 asig*aenv, p6
outs aL, aR

endin

</CsInstruments>
<CsScore>
i1 0 10
e
</CsScore>
</CsSoundSynthesizer>

```

Its output should include a line like this:

```

play instance 0
play instance 1
play instance 2
play instance 3
play instance 4
play instance 5

```

## See Also

*loop\_ge*, *loop\_gt* and *loop\_lt*.

More information on this opcode: [http://www.csoundjournal.com/2006summer/controlFlow\\_part2.html](http://www.csoundjournal.com/2006summer/controlFlow_part2.html), written by Steven Yi. And in the Floss Manuals: [http://en.flossmanuals.net/csound/ch018\\_c-control-structures](http://en.flossmanuals.net/csound/ch018_c-control-structures) [[http://en.flossmanuals.net/csound/ch018\\_c-control-structures](http://en.flossmanuals.net/csound/ch018_c-control-structures)]

## Credits

Istvan Varga.

New in Csound version 5.01

# loop\_lt

loop\_lt — Looping constructions.

## Description

Construction of looping operations.

## Syntax

```
loop_lt indx, incr, imax, label
```

```
loop_lt kndx, kncr, kmax, label
```

## Initialization

*indx* -- i-rate variable to count loop.

*incr* -- value to increment the loop.

*imax* -- maximum value of loop index.

## Performance

*kndx* -- k-rate variable to count loop.

*knrc* -- value to increment the loop.

*kmax* -- maximum value of loop index.

The actions of **loop\_lt** are equivalent to the code

```
indx = indx + incr
if (indx < imax) igoto label
```

or

```
kndx = kndx + knrc
if (kndx < kmax) kgoto label
```

## Examples

Here is an example of the loop\_lt opcode. It uses the file *loop\_lt.csd* [examples/loop\_lt.csd].

### Example 481. Example of the loop\_lt opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
```

```

<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o loop_lt.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

seed 0
gisine ftgen 0, 0, 2^10, 10, 1

instr 1 ;master instrument

ininstr = 5 ;number of called instances
indx = 0
loop:
  prints "play instance %d\\n", indx
  ipan random 0, 1
  ifreq random 100, 1000
  iamp = 1/ininstr
  event_i "i", 10, 0, p3, iamp, ifreq, ipan
  loop_lt indx, 1, ininstr, loop

endin

instr 10

ipeak random 0, 1 ;where is the envelope peak
asig poscil3 p4, p5, gisine
aenv transeg 0, p3*ipeak, 6, 1, p3-p3*ipeak, -6, 0
aL,aR pan2 asig*aenv, p6
outs aL, aR

endin

</CsInstruments>
<CsScore>
i1 0 10
e
</CsScore>
</CsSoundSynthesizer>

```

Its output should include a line like this:

```

play instance 0
play instance 1
play instance 2
play instance 3
play instance 4

```

## See Also

*loop\_ge*, *loop\_gt* and *loop\_le*.

More information on this opcode: [http://www.csoundjournal.com/2006summer/controlFlow\\_part2.html](http://www.csoundjournal.com/2006summer/controlFlow_part2.html), written by Steven Yi. And in the Floss Manuals: [http://en.flossmanuals.net/csound/ch018\\_c-control-structures](http://en.flossmanuals.net/csound/ch018_c-control-structures) [[http://en.flossmanuals.net/csound/ch018\\_c-control-structures](http://en.flossmanuals.net/csound/ch018_c-control-structures)]

## Credits

Istvan Varga.

New in Csound version 5.01

# loopseg

loopseg — Generate control signal consisting of linear segments delimited by two or more specified points.

## Description

Generate control signal consisting of linear segments delimited by two or more specified points. The entire envelope is looped at *kfreq* rate. Each parameter can be varied at k-rate.

## Syntax

```
ksig loopseg kfreq, ktrig, iphase, kvalue0, ktime0 [, kvalue1] [, ktime1] \  
[, kvalue2] [, ktime2][...]
```

## Initialization

*iphase* -- A value between 0 and 1 to say where to start the loop. Zero, the commonest value, indicates the beginning.

## Performance

*ksig* -- Output signal.

*kfreq* -- Repeat rate in Hz or fraction of Hz.

*ktrig* -- If non-zero, retriggers the envelope from start (see *trigger opcode*), before the envelope cycle is completed.

*kvalue0...kvalueN* -- Values of points

*ktime0...ktimeN* -- Times between points; expressed in fractions of a cycle (see below). The final time designates a ramp between the final value and the first value.

*loopseg* opcode is similar to *linseg*, but the entire envelope is looped at *kfreq* rate. Notice that times are not expressed in seconds but in fraction of a cycle. Actually each duration represent is proportional to the other, and the entire cycle duration is proportional to the sum of all duration values.

The sum of all duration is then rescaled according to *kfreq* argument. For example, considering an envelope made up of 3 segments, each segment having 100 as duration value, their sum will be 300. This value represents the total duration of the envelope, and is actually divided into 3 equal parts, a part for each segment.

Actually, the real envelope duration in seconds is determined by *kfreq*. Again, if the envelope is made up of 3 segments, but this time the first and last segments have a duration of 50, whereas the central segment has a duration of 100 again, their sum will be 200. This time 200 represent the total duration of the 3 segments, so the central segment will be twice as long as the other segments.

All parameters can be varied at k-rate. Negative frequency values are allowed, reading the envelope backward. *ktime0* should always be set to 0, except if the user wants some special effect.

## Examples

Here is an example of the loopseg opcode. It uses the file *loopseg.csd* [examples/loopseg.csd].

**Example 482. Example of the loopseg opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o loopseg.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
kfreq init      p4 ; frequency of loop repetition
ifrac =         p5 ; frequency ratio of restart triggers
ktrig metro     kfreq * ifrac ; triggers to restart loop
iphase =        0 ; initial phase
; loop of note values (some glissandi)
;
;                               val dur val dur etc...
knote loopseg kfreq, ktrig, iphase, 40, 1, 40, 0, 43,1,43,0, 49,2,48,0, \
  47,1,47,0, 46,1,46,0, 46,1,47,0, 49,1,49,0, 43,1,43,0, 46,1,46,0, 40,1,39,0
; loop of filter cutoff values (oct format). This loop, half speed of note loop.
kcfoct loopseg kfreq*0.5, ktrig, iphase, 11,2,4,0, 12,1,4,0, 13,1,4,0, \
  11.5,3,4,0, 12.5,1,4,0, 13,2,4,0, 12.5,1,4,0
kenv linseg 0,0.01,1,p3-5.01,1,5,0
ioct =      int((rnd(0.999)*4)-2) ; random value either -1, 0 or 1
asig vco2 0.2*kenv,cpsmidinn(knote)*octave(ioct),0 ; sawtooth
asig moogladder asig,cpsoct(kcfoct),rnd(0.6) ; filter sawtooth
aL,aR pan2 asig,rnd(1) ; random static pan location
outs aL, aR

endin

</CsInstruments>
<CsScore>

; 4 layers, each with a different frequency of loop repetition (p4),
; frequency ratio of restart triggers (p5) and pan position (p6).
i 1 0 30 0.5 [11/19]
i 1 6 30 0.25 [11/13]
i 1 12 30 0.125 [11/16]
i 1 18 30 1 [11/12]
e
</CsScore>
</CsoundSynthesizer>

```

**See Also**

*lpshold loopxseg*

**Credits**

Author: Gabriel Maldonado

New in Version 4.13

# loopsegp

loopsegp — Control signals based on linear segments.

## Description

Generate control signal consisting of linear segments delimited by two or more specified points. The entire envelope can be looped at time-variant rate. Each segment coordinate can also be varied at k-rate.

## Syntax

```
ksig loopsegp kphase, kvalue0, kdur0, kvalue1 \  
    [, kdur1, ... , kdurN-1, kvalueN]
```

## Performance

*ksig* - output signal

*kphase* - point of the sequence read, expressed as a fraction of a cycle (0 to 1)

*kvalue0* ...*kvalueN* - values of points

*kdur0* ...*kdurN-1* - duration of points expressed in fraction of a cycle

*loopsegp* opcode is similar to *loopseg*; the only difference is that, instead of frequency, a time-variant phase is required. If you use *phasor* to get the phase value, you will have a behaviour identical to *loopseg*, but interesting results can be achieved when using phases having non-linear motions, making *loopsegp* more powerful and general than *loopseg*.

## Examples

Here is an example of the *loopsegp* opcode. It uses the file *loopsegp.csd* [examples/loopsegp.csd].

### Example 483. Example of the *loopsegp* opcode.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
-odac      ;;realtime audio out  
;-iadc     ;;uncomment -iadc if realtime audio input is needed too  
; For Non-realtime ouput leave only the line below:  
; -o loopsegp.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>  
  
sr = 44100  
ksmps = 32  
nchnls = 2  
0dbfs = 1  
  
instr 1  
ktrig metro 22/8 ; triggers used to generate new direction values  
kdir trandom ktrig,-2.99,2.99  
kdir = 0.5*int(kdir) ; kdir will either -1, -0.5, 0, 0.5 or 1
```

```
; kphase - looping pointer
kphase phasor kdir
; a loop sequence of midi note numbers and durations
knote loopsegg kphase, 40,1,40,0, 43,1,43,0, 49,2,48,0, \
  47,1,47,0, 46,1,46,0, 46,1,47,0, 49,1,49,0, 43,1,43,0, 46,1,46,0, 40,1,39,0
kmul rspline 0.1,0.8,0.5,5 ; modulation of buzz tone
asig gbuzz 0.2, cpsmidinn(knote), 30, 3, kmul, 1 ; buzz tone
outs asig, asig

schedkwhen ktrig,0,0,2,0,0.1 ; play metronome
endin

instr 2 ; metronome
acps expon 180+rand(40),p3,50
aamp expon 0.05+rand(0.05),p3,0.001
asig poscil aamp-0.001,acps,2
outs asig,asig
endin

</CsInstruments>
<CsScore>
; cosine wave.
f 1 0 16384 11 1
; sine wave.
f 2 0 16384 10 1

i 1 0 360 0.25

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado.

New in Csound 5. (Previously available only on CsoundAV)

# looptseg

`looptseg` — Generate control signal consisting of exponential or linear segments delimited by two or more specified points.

## Description

Generate control signal consisting of controllable exponential segments or linear segments delimited by two or more specified points. The entire envelope is looped at *kfreq* rate. Each parameter can be varied at k-rate.

## Syntax

```
ksig looptseg kfreq, ktrig, iphase, kvalue0, ktype0, ktime0, [, kvalue1] [,ktype1] [, ktime1] \  
    [, kvalue2] [,ktype2] [, ktime2] [...] [, kvalueN] [,ktypeN] [, ktimeN]
```

## Initialization

*iphase* -- A value between 0 and 1 to say where to start the loop. Zero, the commonest value, indicates the beginning.

## Performance

*ksig* -- Output signal.

*kfreq* -- Repeat rate in Hz or fraction of Hz.

*ktrig* -- If non-zero, retriggers the envelope from start (see *trigger opcode*), before the envelope cycle is completed.

*kvalue0...kvalueN* -- Values of points

*ktime0...ktimeN* -- Times between points; expressed in fractions of a cycle (see below). The final time designates a ramp between the final value and the first value.

*ktype0...ktypeN* -- shape of the envelope. If the value is 0 then the shape is linear; otherwise it is a concave exponential (positive type) or a convex exponential (negative type).

*looptseg* opcode is similar to *transeg*, but the entire envelope is looped at *kfreq* rate. Notice that times are not expressed in seconds but in fraction of a cycle. Actually each duration represent is proportional to the other, and the entire cycle duration is proportional to the sum of all duration values.

The sum of all duration is then rescaled according to *kfreq* argument. For example, considering an envelope made up of 3 segments, each segment having 100 as duration value, their sum will be 300. This value represents the total duration of the envelope, and is actually divided into 3 equal parts, a part for each segment.

Actually, the real envelope duration in seconds is determined by *kfreq*. Again, if the envelope is made up of 3 segments, but this time the first and last segments have a duration of 50, whereas the central segment has a duration of 100 again, their sum will be 200. This time 200 represent the total duration of the 3 segments, so the central segment will be twice as long as the other segments.

All parameters can be varied at k-rate. Negative frequency values are allowed, reading the envelope backward. *ktime0* should always be set to 0, except if the user wants some special effect.

## Examples

Here is an example of the *looptseg* opcode. It uses the file *looptseg.csd* [examples/looptseg.csd].

### Example 484. Example of the *looptseg* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o looptseg.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
kfreq = 1          ; frequency of loop repetition
ktrig  = init 0    ; loop restart trigger (not used)
iphase = 0        ; initial phase
ktyp   = line 6,p3,-6 ; explore the useful range of curve types
; loop of filter cutoff values (oct format)
;
; value curve dur.
kcfoct  looptseg kfreq, ktrig, iphase,13, ktyp, 1, \
         4, ktyp, 0, \
         11, ktyp, 1, \
         4

asig  vco2 0.2,cpsmidinn(48),0 ; a sawtooth
asig  moogladder asig,cpsoct(kcfoct),rnd(0.6) ; filter sawtooth
outs  asig, asig

endin

</CsInstruments>
<CsScore>
i 1 0 12
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*lpshold looptseg*

## Credits

Author: John fitch

New in Version 5.12

# loopxseg

loopxseg — Generate control signal consisting of exponential segments delimited by two or more specified points.

## Description

Generate control signal consisting of exponential segments delimited by two or more specified points. The entire envelope is looped at *kfreq* rate. Each parameter can be varied at k-rate.

## Syntax

```
ksig loopxseg kfreq, ktrig, iphase, ktime0, kvalue0 [, ktime1] [, kvalue1] \  
      [, ktime2] [, kvalue2] [...]
```

## Performance

*ksig* -- Output signal.

*kfreq* -- Repeat rate in Hz or fraction of Hz.

*ktrig* -- If non-zero, retriggers the envelope from start (see *trigger opcode*), before the envelope cycle is completed.

*iphase* -- A value between 0 and 1 to say where to start the loop. Zero, the commonest value, indicates the beginning.

*ktime0...ktimeN* -- Times of points; expressed in fraction of a cycle.

*kvalue0...kvalueN* -- Values of points

*loopxseg* opcode is similar to *expseg*, but the entire envelope is looped at *kfreq* rate. Notice that times are not expressed in seconds but in fraction of a cycle. Actually each duration represent is proportional to the other, and the entire cycle duration is proportional to the sum of all duration values.

The sum of all duration is then rescaled according to *kfreq* argument. For example, considering an envelope made up of 3 segments, each segment having 100 as duration value, their sum will be 300. This value represents the total duration of the envelope, and is actually divided into 3 equal parts, a part for each segment.

Actually, the real envelope duration in seconds is determined by *kfreq*. Again, if the envelope is made up of 3 segments, but this time the first and last segments have a duration of 50, whereas the central segment has a duration of 100 again, their sum will be 200. This time 200 represent the total duration of the 3 segments, so the central segment will be twice as long as the other segments.

All parameters can be varied at k-rate. Negative frequency values are allowed, reading the envelope backward. *ktime0* should always be set to 0, except if the user wants some special effect.

## Examples

Here is an example of the *loopxseg* opcode. It uses the file *loopxseg.csd* [examples/loopxseg.csd].

**Example 485. Example of the loopxseg opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o loopxseg.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
kfreq rspline 0.01,20,0.2,1 ; freq. of loop repetition created by random spline
ktrig init 0 ; loop restart trigger (not used)
iphase = 0 ; initial phase
; loop of filter cutoff values (oct format). Rescaled further down.
kcfoct loopxseg kfreq, ktrig, iphase, 1,1,0,0
kenv linseg 0,0.01,1,p3-5.01,1,5,0
asig vco2 0.2*kenv,cpsmidinn(48),0
kdep rspline 5,8,0.2,1 ; filter depth created by a random spline
kcf port cpsoct((kcfoct*kdep)+4), 0.001 ; smooth filter changes
asig moogladder asig,kcf,rnd(0.6)
aL,aR pan2 asig,rnd(1)
outs aL, aR

endin

</CsInstruments>
<CsScore>
i 1 0 60
e
</CsScore>
</CsoundSynthesizer>

```

**See Also**

*lpshold loopseg*

**Credits**

Author: John ffitth

New in Version 5.12

# lorenz

lorenz — Implements the Lorenz system of equations.

## Description

Implements the Lorenz system of equations. The Lorenz system is a chaotic-dynamic system which was originally used to simulate the motion of a particle in convection currents and simplified weather systems. Small differences in initial conditions rapidly lead to diverging values. This is sometimes expressed as the butterfly effect. If a butterfly flaps its wings in Australia, it will have an effect on the weather in Alaska. This system is one of the milestones in the development of chaos theory. It is useful as a chaotic audio source or as a low frequency modulation source.

## Syntax

```
ax, ay, az lorenz ksv, krv, kbv, kh, ix, iy, iz, iskip [, iskipinit]
```

## Initialization

*ix, iy, iz* -- the initial coordinates of the particle.

*iskip* -- used to skip generated values. If *iskip* is set to 5, only every fifth value generated is output. This is useful in generating higher pitched tones.

*iskipinit* (optional, default=0) -- if non zero skip the initialisation of the filter. (New in Csound version 4.23f13 and 5.0)

## Performance

*ksv* -- the Prandtl number or sigma

*krv* -- the Rayleigh number

*kbv* -- the ratio of the length and width of the box in which the convection currents are generated

*kh* -- the step size used in approximating the differential equation. This can be used to control the pitch of the systems. Values of .1-.001 are typical.

The equations are approximated as follows:

$$\begin{aligned}x &= x + h*(s*(y - x)) \\y &= y + h*(-x*z + r*x - y) \\z &= z + h*(x*y - b*z)\end{aligned}$$

The historical values of these parameters are:

```
ks = 10
kr = 28
kb = 8/3
```



## Note

This algorithm uses internal non linear feedback loops which causes audio result to depend on the orchestra sampling rate. For example, if you develop a project with  $sr=48000\text{Hz}$  and if you want to produce an audio CD from it, you should record a file with  $sr=48000\text{Hz}$  and then downsample the file to  $44100\text{Hz}$  using the *srconv* utility.

## Examples

Here is an example of the lorenz opcode. It uses the file *lorenz.csd* [examples/lorenz.csd].

### Example 486. Example of the lorenz opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o lorenz.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 2

; Instrument #1 - a lorenz system in 3D space.
instr 1
; Create a basic tone.
kamp init 25000
kcps init 1000
ifn = 1
asnd oscil kamp, kcps, ifn

; Figure out its X, Y, Z coordinates.
ksv init 10
krv init 28
kbv init 2.667
kh init 0.0003
ix = 0.6
iy = 0.6
iz = 0.6
iskip = 1
ax1, ay1, az1 lorenz ksv, krv, kbv, kh, ix, iy, iz, iskip

; Place the basic tone within 3D space.
kx downsamp ax1
ky downsamp ay1
kz downsamp az1
idist = 1
ift = 0
imode = 1
imdel = 1.018853416
iovr = 2
```

```
aw2, ax2, ay2, az2 spat3d asnd, kx, ky, kz, idist, \  
    ift, imode, imdel, iovr  
  
; Convert the 3D sound to stereo.  
aleft = aw2 + ay2  
aright = aw2 - ay2  
  
outs aleft, aright  
endin  
  
</CsInstruments>  
<CsScore>  
  
; Table #1 a sine wave.  
f 1 0 16384 10 1  
  
; Play Instrument #1 for 5 seconds.  
i 1 0 5  
e  
  
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Author: Hans Mikelson  
February 1999

New in Csound version 3.53

Note added by François Pinot, August 2009

# lorisread

`lorisread` — Imports a set of bandwidth-enhanced partials from a SDIF-format data file, applying control-rate frequency, amplitude, and bandwidth scaling envelopes, and stores the modified partials in memory.

## Syntax

```
lorisread ktimpnt, ifilcod, istoreidx, kfreqenv, kampenv, kbwenv[, ifadetime]
```

## Description

`lorisread` imports a set of bandwidth-enhanced partials from a SDIF-format data file, applying control-rate frequency, amplitude, and bandwidth scaling envelopes, and stores the modified partials in memory.

## Initialization

*ifilcod* - integer or character-string denoting a control-file derived from reassigned bandwidth-enhanced analysis of an audio signal. An integer denotes the suffix of a file `loris.sdif` (e.g. `loris.sdif.1`); a character-string (in double quotes) gives a filename, optionally a full pathname. If not a full pathname, the file is sought first in the current directory, then in the one given by the environment variable `SADIR` (if defined). The reassigned bandwidth-enhanced data file contains breakpoint frequency, amplitude, noisiness, and phase envelope values organized for bandwidth-enhanced additive resynthesis. The control data must conform to one of the SDIF formats that can be

Loris stores partials in SDIF RBEP frames. Each RBEP frame contains one RBEP matrix, and each row in a RBEP matrix describes one breakpoint in a Loris partial. A RBEL frame containing one RBEL matrix describing the labeling of the partials may precede the first RBEP frame in the SDIF file. The RBEP and RBEL frame and matrix definitions are included in the SDIF file's header. In addition to RBEP frames, Loris can also read and write SDIF 1TRC frames. Since 1TRC frames do not represent bandwidth-enhancement or the exact timing of Loris breakpoints, their use is not recommended. 1TRC capabilities are provided to allow interchange with programs that are unable to handle RBEP frames.

*istoreidx*, *ireadidx*, *isrcidx*, *itgtidx* are labels that identify a stored set of bandwidth-enhanced partials. `lorisread` imports partials from a SDIF file and stores them with the integer label `istoreidx`. `lorismorph` morphs sets of partials labeled `isrcidx` and `itgtidx`, and stores the resulting partials with the integer label `istoreidx`. `lorisplay` renders the partials stored with the label `ireadidx`. The labels are used only at initialization time, and may be reused without any cost or benefit in efficiency, and without introducing any interaction between instruments or instances.

*ifadetime* (*optional*) - In general, partials exported from Loris begin and end at non-zero amplitude. In order to prevent artifacts, it is very often necessary to fade the partials in and out, instead of turning them abruptly on and off. Specification of a non-zero `ifadetime` causes partials to fade in at their onsets and to fade out at their terminations. This is achieved by adding two more breakpoints to each partial: one `ifadetime` seconds before the start time and another `ifadetime` seconds after the end time. (However, no breakpoint will be introduced at a time less than zero. If necessary, the onset fade time will be shortened.) The additional breakpoints at the partial onset and termination will have the same frequency and bandwidth as the first and last breakpoints in the partial, respectively, but their amplitudes will be zero. The phase of the new breakpoints will be extrapolated to preserve phase correctness. If no value is specified, `ifadetime` defaults to zero. Note that the `fadetime` may not be exact, since the partial parameter envelopes are sampled at the control rate (`krate`) and indexed by `ktimpnt` (see below), and not by real time.

## Performance

lorisread reads pre-computed Reassigned Bandwidth-Enhanced analysis data from a file stored in SDIF format, as described above. The passage of time through this file is specified by ktimpnt, which represents the time in seconds. ktimpnt must always be positive, but can move forwards or backwards in time, be stationary or discontinuous, as a pointer into the analysis file. kfreqenv is a control-rate transposition factor: a value of 1 incurs no transposition, 1.5 transposes up a perfect fifth, and .5 down an octave. kampenv is a control-rate scale factor that is applied to all partial amplitude envelopes. kbwenv is a control-rate scale factor that is applied to all partial bandwidth or noisiness envelopes. The bandwidth-enhanced partial data is stored in memory with a specified label for future access by another generator.

## Examples

Here is an example of the lorisread opcode. It uses the file *lorisread.csd* [examples/lorisread.csd].

### Example 487. Example of the lorisread opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o lorisread.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

; Play the partials in clarinet.sdif from 0 to 3 sec with 1 ms fadetime
; and no frequency , amplitude, or bandwidth modification.

instr 1

ktime linseg 0, p3, 3 ; linear time function from 0 to 3 seconds
      lorisread ktime, "clarinet.sdif", 1, 1, 1, 1, .001
asig  lorisplay 1, 1, 1, 1
      outs asig, asig
endin

; Play the partials in clarinet.sdif from 0 to 3 sec with 1 ms fadetime
; adding tuning and vibrato, increasing the "breathiness" (noisiness) and overall
; amplitude, and adding a highpass filter.

instr 2

ktime linseg 0, p3, 3 ; linear time function from 0 to 3 seconds
      ; compute frequency scale for tuning
ifscale = cpspch(p4)/cpspch(8.08) ; (original pitch was G#4)
      ; make a vibrato envelope
kvenv linseg 0, p3/6, 0, p3/6, .02, p3/3, .02, p3/6, 0, p3/6, 0
kvib  oscil kvenv, 4, 1 ; table 1, sinusoid
kbwenv linseg 1, p3/6, 1, p3/6, 2, 2*p3/3, 2 ;lots of noise
      lorisread ktime, "clarinet.sdif", 1, 1, 1, 1, .001
```

```
a1    lorisplay 1, ifscale+kvib, 2, kbwenv
asig  atone a1, 1000 ; highpass filter, cutoff 1000 Hz
      outs asig, asig
endin

</CsInstruments>
<CsScore>
; a sinus
f 1 0 4096 10 1

i 1 0 3
i 1 + 1
i 1 + 6
s

; pitch
i 2 1 3 8.08
i 2 3.5 1 8.04
i 2 4 6 8.00
i 2 4 6 8.07
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

This implementation of the Loris unit generators was written by Kelly Fitz ([loris@cerlsoundgroup.org](mailto:loris@cerlsoundgroup.org) [<mailto:loris@cerlsoundgroup.org>]). It is patterned after a prototype implementation of the *lorisplay* unit generator written by Corbin Champion, and based on the method of Bandwidth-Enhanced Additive Synthesis and on the sound morphing algorithms implemented in the Loris library for sound modeling and manipulation. The opcodes were further adapted as a plugin for Csound 5 by Michael Gogins.

# lorismorph

*lorismorph* — Morphs two stored sets of bandwidth-enhanced partials and stores a new set of partials representing the morphed sound. The morph is performed by linearly interpolating the parameter envelopes (frequency, amplitude, and bandwidth, or noisiness) of the bandwidth-enhanced partials according to control-rate frequency, amplitude, and bandwidth morphing functions.

## Syntax

```
lorismorph isrcidx, itgtidx, istoreidx, kfreqmorphenv, kampmorphenv, kbwmorphenv
```

## Description

*lorismorph* morphs two stored sets of bandwidth-enhanced partials and stores a new set of partials representing the morphed sound. The morph is performed by linearly interpolating the parameter envelopes (frequency, amplitude, and bandwidth, or noisiness) of the bandwidth-enhanced partials according to control-rate frequency, amplitude, and bandwidth morphing functions.

## Initialization

*istoreidx*, *ireadidx*, *isrcidx*, *itgtidx* are labels that identify a stored set of bandwidth-enhanced partials. *lorisread* imports partials from a SDIF file and stores them with the integer label *istoreidx*. *lorismorph* morphs sets of partials labeled *isrcidx* and *itgtidx*, and stores the resulting partials with the integer label *istoreidx*. *lorisplay* renders the partials stored with the label *ireadidx*. The labels are used only at initialization time, and may be reused without any cost or benefit in efficiency, and without introducing any interaction between instruments or instances.

## Performance

*lorismorph* generates a set of bandwidth-enhanced partials by morphing two stored sets of partials, the source and target partials, which may have been imported using *lorisread*, or generated by another unit generator, including another instance of *lorismorph*. The morph is performed by interpolating the parameters of corresponding (labeled) partials in the two source sounds. The sound morph is described by three control-rate morphing envelopes. *kfreqmorphenv* describes the interpolation of partial frequency values in the two source sounds. When *kfreqmorphenv* is 0, partial frequencies are obtained from the partials stored at *isrcidx*. When *kfreqmorphenv* is 1, partial frequencies are obtained from the partials at *itgtidx*. When *kfreqmorphenv* is between 0 and 1, the partial frequencies are interpolated between corresponding source and target partials. Interpolation of partial amplitudes and bandwidth (noisiness) coefficients are similarly described by *kampmorphenv* and *kbwmorphenv*.

## Examples

Here is an example of the *lorismorph* opcode. It uses the file *lorismorph.csd* [examples/lorismorph.csd].

### Example 488. Example of the *lorismorph* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o lorismorph.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
; clarinet.sdif and meow.sdif can be found in /manual/examples

; Morph the partials in meow.sdif into the partials in clarinet.sdif over the duration of
; the sustained portion of the two tones (from .2 to 2.0 seconds in the meow, and from
; .5 to 2.1 seconds in the clarinet). The onset and decay portions in the morphed sound are
; specified by parameters p4 and p5, respectively. The morphing time is the time between the
; onset and the decay. The meow partials are shifted in pitch to match the pitch of the clarinet
; tone (D above middle C).

instr 1

ionset = p4
idecay = p5
itmorph = p3 - (ionset + idecay)
ipshift = cpspch(8.02)/cpspch(8.08)

ktme linseg 0, ionset, .2, itmorph, 2.0, idecay, 2.1 ; meow time function, morph from .2 to 2.0 sec
ktcl linseg 0, ionset, .5, itmorph, 2.1, idecay, 2.3 ; clarinet time function, morph from .5 to 2.1
kmurph linseg 0, ionset, 0, itmorph, 1, idecay, 1
  lorisread ktme, "meow.sdif", 1, ipshift, 2, 1, .001
  lorisread ktcl, "clarinet.sdif", 2, 1, 1, 1, .001
  lorismorph 1, 2, 3, kmurph, kmurph, kmurph
asig lorisplay 3, 1, 1, 1
  outs asig, asig
endin

; Morph the partials in clarinet.sdif into the partials in meow.sdif. The start and end times
; for the morph are specified by parameters p4 and p5, respectively. The morph occurs over the
; second of four pitches in each of the sounds, from .75 to 1.2 seconds in the flutter-tongued
; clarinet tone, and from 1.7 to 2.2 seconds in the cat's meow. Different morphing functions are
; used for the frequency and amplitude envelopes, so that the partial amplitudes make a faster
; transition from clarinet to cat than the frequencies. (The bandwidth envelopes use the same
; morphing function as the amplitudes.)

instr 2

ionset = p4
imorph = p5 - p4
irelease = p3 - p5

ktclar linseg 0, ionset, .75, imorph, 1.2, irelease, 2.4
ktmeow linseg 0, ionset, 1.7, imorph, 2.2, irelease, 3.4

kmfreq linseg 0, ionset, 0, .75*imorph, .25, .25*imorph, 1, irelease, 1
kmamp linseg 0, ionset, 0, .75*imorph, .9, .25*imorph, 1, irelease, 1

  lorisread ktclar, "clarinet.sdif", 1, 1, 1, 1, .001
  lorisread ktmeow, "meow.sdif", 2, 1, 1, 1, .001
  lorismorph 1, 2, 3, kmfreq, kmamp, kmamp
asig lorisplay 3, 1, 1, 1
  outs asig, asig

```

```
endin

</CsInstruments>
<CsScore>
;   strt  dur  onset  decay
i 1   0    3   .25   .15
i 1   +    1   .10   .10
i 1   +    6   1.    1.

;   strt  dur  morph_start  morph_end
i 2   9    4   .75         2.75

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

This implementation of the Loris unit generators was written by Kelly Fitz ([loris@cerlsoundgroup.org](mailto:loris@cerlsoundgroup.org) [<mailto:loris@cerlsoundgroup.org>]). It is patterned after a prototype implementation of the *lorisplay* unit generator written by Corbin Champion, and based on the method of Bandwidth-Enhanced Additive Synthesis and on the sound morphing algorithms implemented in the Loris library for sound modeling and manipulation. The opcodes were further adapted as a plugin for Csound 5 by Michael gogins.

# lorisplay

*lorisplay* — renders a stored set of bandwidth-enhanced partials using the method of Bandwidth-Enhanced Additive Synthesis implemented in the Loris software, applying control-rate frequency, amplitude, and bandwidth scaling envelopes.

## Syntax

```
ar lorisplay ireadidx, kfreqenv, kampenv, kbwenv
```

## Description

*lorisplay* renders a stored set of bandwidth-enhanced partials using the method of Bandwidth-Enhanced Additive Synthesis implemented in the Loris software, applying control-rate frequency, amplitude, and bandwidth scaling envelopes.

## Initialization

*istoreidx*, *ireadidx*, *isrcidx*, *itgtidx* are labels that identify a stored set of bandwidth-enhanced partials. *lorisread* imports partials from a SDIF file and stores them with the integer label *istoreidx*. *lorismorph* morphs sets of partials labeled *isrcidx* and *itgtidx*, and stores the resulting partials with the integer label *istoreidx*. *lorisplay* renders the partials stored with the label *ireadidx*. The labels are used only at initialization time, and may be reused without any cost or benefit in efficiency, and without introducing any interaction between instruments or instances.

## Performance

*lorisplay* implements signal reconstruction using Bandwidth-Enhanced Additive Synthesis. The control data is obtained from a stored set of bandwidth-enhanced partials imported from an SDIF file using *lorisread* or constructed by another unit generator such as *lorismorph*. *kfreqenv* is a control-rate transposition factor: a value of 1 incurs no transposition, 1.5 transposes up a perfect fifth, and .5 down an octave. *kampenv* is a control-rate scale factor that is applied to all partial amplitude envelopes. *kbwenv* is a control-rate scale factor that is applied to all partial bandwidth or noisiness envelopes. The bandwidth-enhanced partial data is stored in memory with a specified label for future access by another generator.

## Examples

Here is an example of the *lorisplay* opcode. It uses the file *lorisplay.csd* [examples/lorisplay.csd].

### Example 489. Example of the *lorisplay* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
```

```

; -o lorisplay.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
; clarinet.sdif can be found in /manual/examples

; Play the partials in clarinet.sdif from 0 to 3 sec with 1 ms fadetime
; and no frequency , amplitude, or bandwidth modification.

instr 1

ktime linseg 0, p3, 3.0 ; linear time function from 0 to 3 seconds
lorisread ktime, "clarinet.sdif", 1, 1, 1, 1, .001
kfrq = p4 ; pitch shifting
asig lorisplay 1, kfrq, 1, 1
outs asig, asig
endin

; Play the partials in clarinet.sdif from 0 to 3 sec with 1 ms fadetime
; adding tuning and vibrato, increasing the "breathiness" (noisiness) and overall
; amplitude, and adding a highpass filter.

instr 2

ktime linseg 0, p3, 3.0 ; linear time function from 0 to 3 seconds
; compute frequency scale for tuning

ifscale = cpspch(p4)/cpspch(8.08) ; (original pitch was G#4)
; make a vibrato envelope
kvenv linseg 0, p3/6, 0, p3/6, .02, p3/3, .02, p3/6, 0, p3/6, 0
kvib oscil kvenv, 4, 1 ; table 1, sinusoid
kbwenv linseg 1, p3/6, 1, p3/6, 100, 2*p3/3, 100 ;lots of noise
lorisread ktime, "clarinet.sdif", 1, 1, 1, 1, .001
a1 lorisplay 1, ifscale+kvib, 2, kbwenv
asig atone a1, 1000 ; highpass filter, cutoff 1000 Hz
outs asig, asig
endin

</CsInstruments>
<CsScore>
; a sinusoid
f 1 0 4096 10 1

i 1 0 3 1.2 ; shifted up
i 1 + 1 1.5
i 1 + 6 .5 ; shifted down
s

; strt dur ptch
i 2 1 3 8.08
i 2 3.5 1 8.04
i 2 4 6 8.00
i 2 4 6 8.07
e

</CsScore>
</CsoundSynthesizer>

```

## Credits

This implementation of the Loris unit generators was written by Kelly Fitz (loris@cerlsoundgroup.org [mailto:loris@cerlsoundgroup.org]). It is patterned after a prototype implementation of the *lorisplay* unit

generator written by Corbin Champion, and based on the method of Bandwidth-Enhanced Additive Synthesis and on the sound morphing algorithms implemented in the Loris library for sound modeling and manipulation. The opcodes were further adapted as a plugin for Csound 5 by Michael Gogins.

# loscil

loscil — Read sampled sound from a table.

## Description

Read sampled sound (mono or stereo) from a table, with optional sustain and release looping.

## Syntax

```
ar1 [,ar2] loscil xamp, kcps, ifn [, ibas] [, imod1] [, ibeg1] [, iend1] \  
[, imod2] [, ibeg2] [, iend2]
```

## Initialization

*ifn* -- function table number, typically denoting an sampled sound segment with prescribed looping points loaded using *GEN01*. The source file may be mono or stereo.

*ibas* (optional) -- base frequency in *Hz* of the recorded sound. This optionally overrides the frequency given in the audio file, but is required if the file did not contain one. The default value is 261.626 Hz, i.e. middle C. (New in Csound 4.03). If this value is not known or not present, use 1 here and in *kcps*.

*imod1*, *imod2* (optional, default=-1) -- play modes for the sustain and release loops. A value of 1 denotes normal looping, 2 denotes forward & backward looping, 0 denotes no looping. The default value (-1) will defer to the mode and the looping points given in the source file. Make sure you select an appropriate mode if the file does not contain this information.

*ibeg1*, *iend1*, *ibeg2*, *iend2* (optional, dependent on *mod1*, *mod2*) -- begin and end points of the sustain and release loops. These are measured in *sample frames* from the beginning of the file, so will look the same whether the sound segment is monaural or stereo. If no loop points are specified, and a looping mode (*imod1*, *imod2*) is given, the file will be looped for the whole length.

## Performance

*ar1*, *ar2* -- the output at audio-rate. There is just *ar1* for mono output. However, there is both *ar1* and *ar2* for stereo output.

*xamp* -- the amplitude of the output signal.

*kcps* -- the frequency of the output signal in cycles per second.

*loscil* samples the *fable* audio at a rate determined by *kcps*, then multiplies the result by *xamp*. The sampling increment for *kcps* is dependent on the table's base-note frequency *ibas*, and is automatically adjusted if the orchestra *sr* value differs from that at which the source was recorded. In this unit, *fable* is always sampled with interpolation.

If sampling reaches the *sustain loop* endpoint and looping is in effect, the point of sampling will be modified and *loscil* will continue reading from within that loop segment. Once the instrument has received a *turnoff* signal (from the score or from a MIDI noteoff event), the next sustain endpoint encountered will be ignored and sampling will continue towards the *release loop* end-point, or towards the last sample (henceforth to zeros).

*loscil* is the basic unit for building a sampling synthesizer. Given a sufficient set of recorded piano tones, for example, this unit can resample them to simulate the missing tones. Locating the sound source nearest a desired pitch can be done via table lookup. Once a sampling instrument has begun, its *turnoff* point may be unpredictable and require an external *release* envelope; this is often done by gating the sampled audio with *linenr*, which will extend the duration of a turned-off instrument by a specific period while it implements a decay.

If you want to loop the whole file, specify a looping mode in *imodl* and do not enter any values for *ibeg* and *iend*.



### Note to Windows users

Windows users typically use back-slashes, “\”, when specifying the paths of their files. As an example, a Windows user might use the path “c:\music\samples\loop001.wav”. This is problematic because back-slashes are normally used to specify special characters.

To correctly specify this path in Csound, one may alternately:

- Use forward slashes: c:/music/samples/loop001.wav
- Use back-slash special characters, “\\”: c:\\music\\samples\\loop001.wav



### Note

This is mono *loscil*:

```
a1 loscil 10000, 1, 1, 1 ,1
```

...and this is stereo *loscil*:

```
a1, a2 loscil 10000, 1, 1, 1 ,1
```

## Examples

Here is an example of the *loscil* opcode. It uses the files *loscil.csd* [examples/loscil.csd], *mary.wav* [examples/mary.wav] and *kickroll.wav* [examples/kickroll.wav].

### Example 490. Example of the *loscil* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o loscil.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
```

```
instr 1

ichnls = fchnls(p4)
print ichnls

if (ichnls == 1) then
  asigL loscil .8, 1, p4, 1
  asigR = asigL
elseif (ichnls == 2) then
  asigL, asigR loscil .8, 1, p4, 1
; safety precaution if not mono or stereo
else
  asigL = 0
  asigR = 0
endif
  outs asigL, asigR

endin
</CsInstruments>
<CsScore>
f 1 0 0 1 "mary.wav" 0 0 0
f 2 0 0 1 "kickroll.wav" 0 0 0

i 1 0 3 1 ;mono file
i 1 + 2 2 ;stereo file
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*loscil3* and *GEN01*

## Credits

Note about the mono/stereo difference was contributed by Rasmus Ekman.

# loscil3

loscil3 — Read sampled sound from a table using cubic interpolation.

## Description

Read sampled sound (mono or stereo) from a table, with optional sustain and release looping, using cubic interpolation.

## Syntax

```
ar1 [,ar2] loscil3 xamp, kcps, ifn [, ibas] [, imod1] [, ibeg1] [, iend1] \  
[, imod2] [, ibeg2] [, iend2]
```

## Initialization

*ifn* -- function table number, typically denoting an sampled sound segment with prescribed looping points loaded using *GEN01*. The source file may be mono or stereo.

*ibas* (optional) -- base frequency in *Hz* of the recorded sound. This optionally overrides the frequency given in the audio file, but is required if the file did not contain one. The default value is 261.626 Hz, i.e. middle C. (New in Csound 4.03). If this value is not known or not present, use 1 here and in *kcps*.

*imod1*, *imod2* (optional, default=-1) -- play modes for the sustain and release loops. A value of 1 denotes normal looping, 2 denotes forward & backward looping, 0 denotes no looping. The default value (-1) will defer to the mode and the looping points given in the source file. Make sure you select an appropriate mode if the file does not contain this information.

*ibeg1*, *iend1*, *ibeg2*, *iend2* (optional, dependent on *mod1*, *mod2*) -- begin and end points of the sustain and release loops. These are measured in *sample frames* from the beginning of the file, so will look the same whether the sound segment is monaural or stereo. If no loop points are specified, and a looping mode (*imod1*, *imod2*) is given, the file will be looped for the whole length.

## Performance

*ar1*, *ar2* -- the output at audio-rate. There is just *ar1* for mono output. However, there is both *ar1* and *ar2* for stereo output.

*xamp* -- the amplitude of the output signal.

*kcps* -- the frequency of the output signal in cycles per second.

*loscil3* is identical to *loscil* except that it uses cubic interpolation. New in Csound version 3.50.



### Note to Windows users

Windows users typically use back-slashes, “\”, when specifying the paths of their files. As an example, a Windows user might use the path “c:\music\samples\loop001.wav”. This is problematic because back-slashes are normally used to specify special characters.

To correctly specify this path in Csound, one may alternately:

- Use forward slashes: `c:/music/samples/loop001.wav`
- Use back-slash special characters, “\\”: `c:\\music\\samples\\loop001.wav`



## Note

This is mono loscil3:

```
a1 loscil3 10000, 1, 1, 1, 1
```

...and this is stereo loscil3:

```
a1, a2 loscil3 10000, 1, 1, 1, 1
```

## Examples

Here is an example of the `loscil3` opcode. It uses the file `loscil3.csd` [examples/loscil3.csd], `mary.wav` [examples/mary.wav] and `kickroll.wav` [examples/kickroll.wav].

### Example 491. Example of the `loscil3` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o loscil3.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ichnls = fchnls(p4)
print ichnls

if (ichnls == 1) then
  asigL loscil3 .8, 1, p4, 1
  asigR = asigL
elseif (ichnls == 2) then
  asigL, asigR loscil3 .8, 1, p4, 1
; safety precaution if not mono or stereo
else
  asigL = 0
  asigR = 0
endif
outs asigL, asigR

endin
</CsInstruments>
<CsScore>
```

```
f 1 0 0 1 "mary.wav" 0 0 0
f 2 0 0 1 "kickroll.wav" 0 0 0

i 1 0 3 1 ;mono file
i 1 + 2 2 ;stereo file
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*loscil* and *GENO1*

## Credits

Note about the mono/stereo difference was contributed by Rasmus Ekman.

# loscilx

loscilx — Read multi-channel sampled sound from a table.

## Description

Read sampled sound (up to 16 channels) from a table, with optional sustain and release looping.

## Syntax

```
ar1 [, ar2, ar3, ar4, ar5, ar6, ar7, ar8, ar9, ar10, ar11, ar12, ar13, ar14, \
    ar15, ar16] loscilx xamp, kcps, ifn \
    [, iwsiz, ibas, istr, imod, ibeg, iend]
```

## Initialization

*ifn* -- function table number, typically denoting an sampled sound segment with prescribed looping points loaded using *GEN01*. The source file may have up to 16 channels.

*iwsiz* (optional) -- window size used in interpolation. *iwsiz* (optional, defaults to zero) -- interpolation window size, in samples. Can be one of the following:

- 1: round to nearest sample (no interpolation, for *kpitch*=1)
- 2: linear interpolation
- 4: cubic interpolation
- >= 8: *iwsiz* point sinc interpolation with anti-aliasing (slow)

Zero or negative values select the default, which is cubic interpolation.

*ibas* (optional) -- base frequency in *Hz* of the recorded sound. This optionally overrides the frequency given in the audio file, but is required if the file did not contain one. The default value is 261.626 Hz, i.e. middle C. (New in Csound 4.03). If this value is not known or not present, use 1 here and in *kcps*.

*istr* (optional, default 0) -- Frame to start reading the data. If this is not an integer the the data is interpolated (see *iwsiz*).

*imod* (optional, default -1) -- play mode for the sustain and release loops. A value of 1 denotes normal looping, 2 denotes forward & backward looping, 0 denotes no looping. The default value (-1) will defer to the mode and the looping points given in the source file. Make sure you select an appropriate mode if the file does not contain this information.

*ibeg*, *iend* (optional, depending on *imod*) -- begin and end points of the sustain and release loops. These are measured in *sample frames* from the beginning of the file. If no loop points are specified, and a looping mode (*imod* is given, the file will be looped for the whole length.

## Performance

*ar1*, *ar2*, ... -- the output at audio-rate. The number of outputs must match the number of channels in the sample file.

*xamp* -- the amplitude of the output signal.

*kcps* -- the factor to read the file. For example, a value of 1 has no pitch change, 1.5 is up a fifth and 2 an octave.

*loscilx* samples the *fable* audio at a rate determined by *kcps*, then multiplies the result by *xamp*. The sampling increment for *kcps* is dependent on the table's base-note frequency *ibas*, and is automatically adjusted if the orchestra *sr* value differs from that at which the source was recorded. In this unit, *fable* is always sampled with interpolation.

If sampling reaches the *sustain loop* endpoint and looping is in effect, the point of sampling will be modified and *loscil* will continue reading from within that loop segment. Once the instrument has received a *turnoff* signal (from the score or from a MIDI noteoff event), the next sustain endpoint encountered will be ignored and sampling will continue towards the *release loop* end-point, or towards the last sample (henceforth to zeros).

If you want to loop the whole file, specify a looping mode in *imod* and do not enter any values for *ibeg* and *iend*.

## Examples

Here is an example of the *loscilx* opcode. It uses the files *loscil.csd* [examples/loscil.csd] and *kickroll.wav* [examples/kickroll.wav].

### Example 492. Example of the *loscilx* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o loscil.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 44100
ksmps = 1
nchnls = 2
0dbfs = 1.0

gS_loop = "kickroll.wav"
gisr filesr gS_loop
gilength filelen gS_loop
gibeats = 16
gi_sampleleft ftgen 1, 0, 0, 1, gS_loop, 0, 4, 0

instr 1
  idur = p3
  iamp = p4
  ibeat = p5
  itune = p6
  ipos = ibeat / gibeats * gilength * gisr

  aenv linseg iamp, idur - 0.01, iamp, 0.01, 0
  a1, a2 loscilx aenv, itune, 1, 0, 1, ipos, 0
```

```
    outs a1, a2
  endin

</CsInstruments>
<CsScore>

t 0 170

; Measure 1
i 1 0.0 0.5 0.707 0 1
i 1 1.0 0.5 0.707 1 1
i 1 2.5 0.5 0.707 0 1
i 1 3.0 0.5 0.707 1 1

; Measure 2
i 1 4.0 0.5 0.707 0 1
i 1 5.0 0.5 0.707 1 1
i 1 6.5 0.5 0.707 0 1
i 1 7.0 0.5 0.707 1 1

; Measure 3
i 1 8.0 0.5 0.707 0 1
i 1 9.0 0.5 0.707 1 1
i 1 10.5 0.5 0.707 0 1
i 1 11.0 0.5 0.707 1 1

; Measure 4
i 1 12.0 0.5 0.707 0 1
i 1 13.0 0.5 0.707 1 1
i 1 14.5 0.5 0.707 0 1
i 1 15.0 0.5 0.707 1 1

</CsScore>
</CsoundSynthesizer>
```

## See Also

*loscil*

## Credits

Written by Istvan Varga.

2006

New in Csound 5.03

# lowpass2

lowpass2 — A resonant lowpass filter.

## Description

Implementation of a resonant second-order lowpass filter.

## Syntax

```
ares lowpass2 asig, kcf, kq [, iskip]
```

## Initialization

*iskip* -- initial disposition of internal data space. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*asig* -- input signal to be filtered

*kcf* -- cutoff or resonant frequency of the filter, measured in Hz

*kq* -- Q of the filter, defined, for bandpass filters, as bandwidth/cutoff. *kq* should be between 1 and 500

*lowpass2* is a second order IIR lowpass filter, with k-rate controls for cutoff frequency (*kcf*) and Q (*kq*). As *kq* is increased, a resonant peak forms around the cutoff frequency, transforming the lowpass filter response into a response that is similar to a bandpass filter, but with more low frequency energy. This corresponds to an increase in the magnitude and "sharpness" of the resonant peak. For high values of *kq*, a scaling function such as *balance* may be required. In practice, this allows for the simulation of the voltage-controlled filters of analog synthesizers, or for the creation of a pitch of constant amplitude while filtering white noise.

## Examples

Here is an example of the lowpass2 opcode. It uses the file *lowpass2.csd* [examples/lowpass2.csd].

### Example 493. Example of the lowpass2 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc      -d          ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o lowpass2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```
/* Written by Sean Costello */
; Orchestra file for resonant filter sweep of a sawtooth-like waveform.
sr = 44100
kr = 2205
ksmps = 20
nchnls = 1

instr 1

idur = p3
ifreq = p4
iamp = p5 * .5
iharms = (sr*.4) / ifreq

; Sawtooth-like waveform
asig gbuzz 1, ifreq, iharms, 1, .9, 1

; Envelope to control filter cutoff
kfreq linseg 1, idur * 0.5, 5000, idur * 0.5, 1

afilt lowpass2 asig,kfreq, 30

; Simple amplitude envelope
kenv linseg 0, .1, iamp, idur -.2, iamp, .1, 0
out afilt * kenv

endin

</CsInstruments>
<CsScore>

/* Written by Sean Costello */
f1 0 8192 9 1 1 .25

i1 0 5 100 1000
i1 5 5 200 1000
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Sean Costello  
Seattle, Washington  
August 1999

New in Csound version 4.0

# lowres

lowres — Another resonant lowpass filter.

## Description

*lowres* is a resonant lowpass filter.

## Syntax

```
ares lowres asig, kcutoff, kresonance [, iskip]
```

## Initialization

*iskip* -- initial disposition of internal data space. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*asig* -- input signal

*kcutoff* -- filter cutoff frequency point

*kresonance* -- resonance amount

*lowres* is a resonant lowpass filter derived from a Hans Mikelson orchestra. This implementation is much faster than implementing it in Csound language, and it allows *kr* lower than *sr*. *kcutoff* is not in Hz and *kresonance* is not in dB, so experiment for the finding best results.

## Examples

Here is an example of the *lowres* opcode. It uses the file *lowres.csd* [examples/lowres.csd].

### Example 494. Example of the *lowres* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o lowres.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
```

```
instr 1

kres = p4
asig vco .2, 220, 1 ;sawtooth

kcut line 1000, p3, 10 ;note: kcut is not in Hz
as lowres asig, kcut, kres ;note: kres is not in dB
aout balance as, asig ;avoid very loud sounds
outs aout, aout

endin
</CsInstruments>
<CsScore>
; a sine
f 1 0 16384 10 1

i 1 0 4 3
i 1 + 4 30
i 1 + 4 60
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*lowresx*

## Credits

Author: Gabriel Maldonado (adapted by John ffitich)  
Italy

New in Csound version 3.49

# lowresx

lowresx — Simulates layers of serially connected resonant lowpass filters.

## Description

*lowresx* is equivalent to more layers of *lowres* with the same arguments serially connected.

## Syntax

```
ares lowresx asig, xcutoff, xresonance [, inumlayer] [, iskip]
```

## Initialization

*inumlayer* -- number of elements in a *lowresx* stack. Default value is 4. There is no maximum.

*iskip* -- initial disposition of internal data space. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*asig* -- input signal

*xcutoff* -- filter cutoff frequency point

*xresonance* -- resonance amount

*lowresx* is equivalent to more layer of *lowres* with the same arguments serially connected. Using a stack of a larger number of filters allows a sharper cutoff. This is faster than using a larger number of instances of *lowres* in a Csound orchestra because only one initialization and k cycle are needed at time and the audio loop falls entirely inside the cache memory of processor. Based on an orchestra by Hans Mikelson

## Examples

Here is an example of the lowresx opcode. It uses the file *lowresx.csd* [examples/lowresx.csd].

### Example 495. Example of the lowresx opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o lowresx.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
```

```
nchnls = 2
odbfs = 1

instr 1

kres = p4
inumlayer = 4

kenv linseg 0, p3*.1, 1, p3*.8, 1, p3*.1, 0 ;envelope
asig vco .3 * kenv, 220, 1 ;sawtooth
kcut line 30, p3, 1000 ;note: kcut is not in Hz
alx lowresx asig, kcut, kres, inumlayer ;note: kres is not in dB
aout balance alx, asig ;avoid very loud sounds
outs aout, aout

endin
</CsInstruments>
<CsScore>
; sine wave
f 1 0 16384 10 1

i 1 0 5 1
i 1 + 5 3
i 1 + 5 20
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*lowres*

## Credits

Author: Gabriel Maldonado (adapted by John ffitch)  
Italy

New in Csound version 3.49

Audio rate parameters introduced in version 6.02

November 2013.

# lpf18

lpf18 — A 3-pole sweepable resonant lowpass filter.

## Description

Implementation of a 3 pole sweepable resonant lowpass filter.

## Syntax

```
ares lpf18 asig, xfco, xres, xdist [, iskip]
```

## Initialization

*iskip* (optional, default=0) -- Skip initialization if present and non-zero.

## Performance

*xfco* -- the filter cutoff frequency in Hz. Should be in the range 0 to  $sr/2$ .

*xres* -- the amount of resonance. Self-oscillation occurs when *xres* is approximately 1. Should usually be in the range 0 to 1, however, values slightly greater than 1 are possible for more sustained oscillation and an “overdrive” effect.

*xdist* -- amount of distortion.  $kdist = 0$  gives a clean output.  $xdist > 0$  adds *tanh()* distortion controlled by the filter parameters, in such a way that both low cutoff and high resonance increase the distortion amount. Some experimentation is encouraged.

*lpf18* is a digital emulation of a 3 pole (18 dB/oct.) lowpass filter capable of self-oscillation with a built-in distortion unit. It is really a 3-pole version of *moogvcf*, retuned, recalibrated and with some performance improvements. The tuning and feedback tables use no more than 6 adds and 6 multiplies per control rate. The distortion unit, itself, is based on a modified *tanh* function driven by the filter controls.



### Note

Before version 6.04 this filter requires that the input signal be normalized to one.

## Examples

Here is an example of the lpf18 opcode. It uses the file *lpf18.csd* [examples/lpf18.csd].

### Example 496. Example of the lpf18 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d          ;;RT audio I/O
```

```
; For Non-realtime ouput leave only the line below:
; -o lpf18.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Generate a sine waveform.
  ; Note that its amplitude (kamp) ranges from 0 to 1.
  kamp init 1
  kcps init 440
  knh init 3
  ifn = 1
  asine buzz kamp, kcps, knh, ifn

  ; Filter the sine waveform.
  ; Vary the cutoff frequency (kfco) from 300 to 3,000 Hz.
  kfco line 300, p3, 3000
  kres init 0.8
  kdist = p4
  ivol = p5
  aout lpf18 asine, kfco, kres, kdist

  out aout * ivol
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; different distortion and volumes to compensate
i 1 0 4 0.2 30000
i 1 4.5 4 0.9 27000
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Josep M Comajuncosas  
Spain  
December 2000

Example written by Kevin Conder with help from Iain Duncan. Thanks goes to Iain for helping with the example.

New in Csound version 4.10

Audio rate parameters introduced in version 6.02

October 2013.

# lpfreson

lpfreson — Resynthesises a signal from the data passed internally by a previous lpread, applying formant shifting.

## Description

Resynthesises a signal from the data passed internally by a previous *lpread*, applying formant shifting.

## Syntax

```
ares lpfreson asig, kfrqratio
```

## Performance

*asig* -- an audio driving function for resynthesis.

*kfrqratio* -- frequency ratio. Must be greater than 0.

*lpfreson* receives values internally produced by a leading *lpread*. *lpread* gets its values from the control file according to the input value *ktimpnt* (in seconds). If *ktimpnt* proceeds at the analysis rate, time-normal synthesis will result; proceeding at a faster, slower, or variable rate will result in time-warped synthesis. At each k-period, *lpread* interpolates between adjacent frames to more accurately determine the parameter values (presented as output) and the filter coefficient settings (passed internally to a subsequent *lpreson*).

The error signal *kerr* (between 0 and 1) derived during predictive analysis reflects the deterministic/random nature of the analyzed source. This will emerge low for pitched (periodic) material and higher for noisy material. The transition from voiced to unvoiced speech, for example, produces an error signal value of about .3. During synthesis, the error signal value can be used to determine the nature of the *lpreson* driving function: for example, by arbitrating between pitched and non-pitched input, or even by determining a mix of the two. In normal speech resynthesis, the pitched input to *lpreson* is a wideband periodic signal or pulse train derived from a unit such as *buzz*, and the nonpitched source is usually derived from *rand*. However, any audio signal can be used as the driving function, the only assumption of the analysis being that it has a flat response.

*lpfreson* is a formant shifted *lpreson*, in which *kfrqratio* is the (cps) ratio of shifted to original formant positions. This permits synthesis in which the source object changes its apparent acoustic size. *lpfreson* with *kfrqratio* = 1 is equivalent to *lpreson*.

Generally, *lpreson* provides a means whereby the time-varying content and spectral shaping of a composite audio signal can be controlled by the dynamic spectral content of another. There can be any number of *lpread/lpreson* (or *lpfreson*) pairs in an instrument or in an orchestra; they can read from the same or different control files independently.

## Examples

Here is an example of the lpfreson opcode. It uses the file *lpfreson.csd* [examples/lpfreson.csd].

### Example 497. Example of the lpfreson opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o lpfreson.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
;do not use -a option when analyzing "fox.wav" with lpfreson,
;it needs a filter coefficient type of file
instr 1

ilen filelen "fox.wav" ; length of soundfile
prints "fox.wav = %f seconds\\n",ilen

ktime line 0, p3, p4
krmsr,krms0,kerr,kcps lpread ktime,"fox_nopoles.lpc"
krms0 = krms0*.00001 ; low volume
asig buzz krms0, kcps, int(sr/2/kcps), 1 ; max harmonics without aliasing
aout lpfreson asig, 1.2
asig clip aout, 0, 1 ; prevents distortion
outs asig, asig

endin
</CsInstruments>
<CsScore>
; sine
f1 0 4096 10 1

i 1 0 2.8 1 ; first words only
i 1 4 2.8 2.8 ; whole sentence
e
</CsScore>
</CsoundSynthesizer>
```

The audio file “fox.wav” is 2.8 seconds long. So *filelen*'s output should be a line like this:

```
fox.wav = 2.756667 seconds
```

## See Also

*lpread*, *lpreson*

# lphasor

lphasor — Generates a table index for sample playback

## Description

This opcode can be used to generate table index for sample playback (e.g. tablexkt).

## Syntax

```
ares lphasor xtrns [, ilps] [, ilpe] [, imode] [, istrtr] [, istor]
```

## Initialization

*ilps* -- loop start.

*ilpe* -- loop end (must be greater than *ilps* to enable looping). The default value of *ilps* and *ilpe* is zero.

*imode* (optional: default = 0) -- loop mode. Allowed values are:

- 0: no loop
- 1: forward loop
- 2: backward loop
- 3: forward-backward loop

*istrtr* (optional: default = 0) -- The initial output value (phase). It must be less than *ilpe* if looping is enabled, but is allowed to be greater than *ilps* (i.e. you can start playback in the middle of the loop).

*istor* (optional: default = 0) -- skip initialization if set to any non-zero value.

## Performance

*ares* -- a raw table index in samples (same unit for loop points). Can be used as index with the table opcodes.

*xtrns* -- transpose factor, expressed as a playback ratio. *ares* is incremented by this value, and wraps around loop points. For example, 1.5 means a fifth above, 0.75 means fourth below. It is not allowed to be negative.

## Examples

Here is an example of the lphasor opcode. It uses the file *lphasor.csd* [examples/lphasor.csd].

### Example 498. Example of the lphasor opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform
```

```
; Audio out   Audio in
-odac         -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o lphashor.wav -W ;; for file output any platform
</CsOptions>

<CsInstruments>
; Example by Jonathan Murphy Dec 2006

  sr      = 44100
  ksmps   = 10
  nchnls  = 1

  instr 1

  ifn     = 1 ; table number
  ilen    = nsamp(ifn) ; return actual number of samples in table
  itrns   = 1 ; no transposition
  ilps    = 0 ; loop starts at index 0
  ilpe    = ilen ; ends at value returned by nsamp above
  imode   = 3 ; loop forwards & backwards
  istrt   = 10000 ; commence playback at index 10000 samples
; lphasor provides index into f1
  alphas  lphasor itrns, ilps, ilpe, imode, istrt
  atab    tablei  alphas, ifn
; amplify signal
  atab    = atab * 10000

  out     atab

  endin

</CsInstruments>

<CsScore>
f 1 0 262144 1 "beats.wav" 0 4 1
i1 0 60
e
</CsScore>

</CsoundSynthesizer>
```

## Credits

Author: Istvan Varga

January 2002

Example by: Jonathan Murphy

New in version 4.18

Updated April 2002 and November 2002 by Istvan Varga

# lpinterp

lpslot, lpinterp — Computes a new set of poles from the interpolation between two analysis.

## Description

Computes a new set of poles from the interpolation between two analysis.

## Syntax

```
lpinterp islot1, islot2, kmix
```

## Initialization

*islot1* -- slot where the first analysis was stored

*islot2* -- slot where the second analysis was stored

*kmix* -- mix value between the two analysis. Should be between 0 and 1. 0 means analysis 1 only. 1 means analysis 2 only. Any value in between will produce interpolation between the filters.

*lpinterp* computes a new set of poles from the interpolation between two analysis. The poles will be stored in the current *lpslot* and used by the next *lpreson* opcode.

## Examples

Here is a typical orc using the opcodes:

```
ipower init 50000 ; Define sound generator
ifreq  init 440
asrc  buzz ipower, ifreq, 10, 1

ktime  line 0, p3, p3          ; Define time lin
      lpslot 0                ; Read square data poles
krmsr, krmso, kerr, kcps lpread ktime, "square.pol"
      lpslot 1                ; Read triangle data poles
krmsr, krmso, kerr, kcps lpread ktime, "triangle.pol"
kmix   line 0, p3, 1          ; Compute result of mixing
      lpinterp 0, 1, kmix     ; and balance power
ares  lpreson asrc
aout  balance ares, asrc
      out aout
```

## See Also

*lpslot*

## Credits

Author: Gabriel Maldonado

# lposcil

lposcil — Read sampled sound from a table with looping and high precision.

## Description

Read sampled sound (mono or stereo) from a table, with looping, and high precision.

## Syntax

```
ares lposcil kamp, kfreqratio, kloop, kend, ifn [, iphs]
```

## Initialization

*ifn* -- function table number

## Performance

*kamp* -- amplitude

*kfreqratio* -- multiply factor of table frequency (for example: 1 = original frequency, 1.5 = a fifth up, .5 = an octave down)

*kloop* -- start loop point (in samples)

*kend* -- end loop point (in samples)

*lposcil* (looping precise oscillator) allows varying at k-rate, the starting and ending point of a sample contained in a table (*GEN01*). This can be useful when reading a sampled loop of a wavetable, where repeat speed can be varied during the performance.

## Examples

Here is an example of the lposcil opcode. It uses the file *lposcil.csd* [examples/lposcil.csd].

### Example 499. Example of the lposcil opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o lposcil.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
```

```
Odbfs = 1

instr 1

kcps = 1.5 ; a fifth up
kloop = 0 ; loop start time in samples
kend = 45000 ; loop end time in samples

asig lposcil 1, kcps, kloop, kend, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
; Its table size is deferred,
; and format taken from the soundfile header.
f 1 0 0 1 "beats.wav" 0 0 0

; Play Instrument #1 for 6 seconds.
; This will loop the drum pattern several times.
i 1 0 6

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*lposcil3, lposcila, lposcilsa, lposcilsa2*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.52

# lposcil3

lposcil3 — Read sampled sound from a table with high precision and cubic interpolation.

## Description

Read sampled sound (mono or stereo) from a table, with looping, and high precision. *lposcil3* uses cubic interpolation.

## Syntax

```
ares lposcil3 kamp, kfreqratio, kloop, kend, ifn [, iphs]
```

## Initialization

*ifn* -- function table number

## Performance

*kamp* -- amplitude

*kfreqratio* -- multiply factor of table frequency (for example: 1 = original frequency, 1.5 = a fifth up, .5 = an octave down)

*kloop* -- start loop point (in samples)

*kend* -- end loop point (in samples)

*lposcil3* (looping precise oscillator) allows varying at k-rate, the starting and ending point of a sample contained in a table (*GEN01*). This can be useful when reading a sampled loop of a wavetable, where repeat speed can be varied during the performance.

## Examples

Here is an example of the *lposcil3* opcode. It uses the file *lposcil3.csd* [examples/lposcil3.csd].

### Example 500. Example of the *lposcil3* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o lposcil3.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
```

```
nchnls = 2
nbfbs = 1

instr 1

kcps = 1.5 ; a fifth up
kloop = 0 ; loop start time (in samples)
kend line 45000, p3, 10000 ; vary loop end time (in samples)

asig lposcil3 1, kcps, kloop, kend, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
; Its table size is deferred,
; and format taken from the soundfile header.
f 1 0 0 1 "beats.wav" 0 0 0

; Play Instrument #1 for 6 seconds.
; This will loop the drum pattern several times.
i 1 0 6

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*lposcil*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.52

# lposcila

*lposcila* — Read sampled sound from a table with looping and high precision.

## Description

*lposcila* reads sampled sound from a table with looping and high precision.

## Syntax

```
ar lposcila aamp, kfreqratio, kloop, kend, ift [,iphs]
```

## Initialization

*ift* -- function table number

*iphs* -- initial phase (in samples)

## Performance

*ar* -- output signal

*aamp* -- amplitude

*kfreqratio* -- multiply factor of table frequency (for example: 1 = original frequency, 1.5 = a fifth up, .5 = an octave down)

*kloop* -- start loop point (in samples)

*kend* -- end loop point (in samples)

*lposcila* is the same as *lposcil*, but has an audio-rate amplitude argument (instead of k-rate) to allow fast envelope transients.

## Examples

Here is an example of the *lposcila* opcode. It uses the file *lposcila.csd* [examples/lposcila.csd].

### Example 501. Example of the *lposcila* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o lposcila.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kcps = 1.3      ;a 3d up
kloop = 0      ;loop start time in samples
kend = 10000   ;loop end time in samples

aenv expsega 0.01, 0.1, 1, 0.1, 0.5, 0.5, 0.01 ;envelope with fast and short segment
asig lposcila aenv, kcps, kloop, kend, 1 ;use it for amplitude
outs asig, asig

endin
</CsInstruments>
<CsScore>
; Its table size is deferred,
; and format taken from the soundfile header.
f 1 0 0 1 "beats.wav" 0 0 0

; Play Instrument #1 for 6 seconds.
; This will loop the drum pattern several times.
i 1 0 2

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*lposcil, lposcilsa, lposcilsa2*

## Credits

Author: Gabriel Maldonado

New in version 5.06

# lposcilsa

lposcilsa — Read stereo sampled sound from a table with looping and high precision.

## Description

*lposcilsa* reads stereo sampled sound from a table with looping and high precision.

## Syntax

```
ar1, ar2 lposcilsa aamp, kfreqratio, kloop, kend, ift [,iphs]
```

## Initialization

*ift* -- function table number

*iphs* -- initial phase (in samples)

## Performance

*ar1*, *ar2* -- output signal

*aamp* -- amplitude

*kfreqratio* -- multiply factor of table frequency (for example: 1 = original frequency, 1.5 = a fifth up, .5 = an octave down)

*kloop* -- start loop point (in samples)

*kend* -- end loop point (in samples)

*lposcilsa* is the same as *lposcila*, but works with stereo files loaded with *GEN01*.

## Examples

Here is an example of the *lposcilsa* opcode. It uses the file *lposcilsa.csd* [examples/lposcilsa.csd].

### Example 502. Example of the *lposcilsa* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o lposcilsa.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
```

```
ksmps = 32
nchnls = 2
Odbfs = 1

instr 1

kcps = 1.3      ;a 3th up
kloop = 0      ;loop start time in samples
kend = 45000   ;loop end time in samples

aenv expsega 0.01, 0.1, 1, 0.1, 0.5, 0.5, 0.01 ;envelope with fast and short segment
aL, aR lposcilsa aenv, kcps, kloop, kend, 1 ;use it for amplitude
outs aL, aR

endin
</CsInstruments>
<CsScore>
; table size of stereo file is deferred,
; and format taken from the soundfile header.
f 1 0 0 1 "kickroll.wav" 0 0 0

; Play Instrument #1 for 6 seconds.
; This will loop the drum pattern several times.
i 1 0 2

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*lposcil, lposcila, lposcilsa2*

## Credits

Author: Gabriel Maldonado

New in version 5.06

# lposcilsa2

lposcilsa2 — Read stereo sampled sound from a table with looping and high precision.

## Description

*lposcilsa2* reads stereo sampled sound from a table with looping and high precision.

## Syntax

```
ar1, ar2 lposcilsa2 aamp, kfreqratio, kloop, kend, ift [,iphs]
```

## Initialization

*ift* -- function table number

*iphs* -- initial phase (in samples)

## Performance

*ar1*, *ar2* -- output signal

*aamp* -- amplitude

*kfreqratio* -- multiply factor of table frequency (for example: 1 = original frequency, 2 = an octave up). Only integers are allowed

*kloop* -- start loop point (in samples)

*kend* -- end loop point (in samples)

*lposcilsa2* is the same as *lposcilsa*, but no interpolation is implemented and only works with integer *kfreqratio* values. Much faster than *lposcilsa*, it is mainly intended to be used with *kfreqratio* = 1, being in this case a fast substitute of *soundin*, since the soundfile must be entirely loaded in memory.

## Examples

Here is an example of the *lposcilsa2* opcode. It uses the file *lposcilsa2.csd* [examples/lposcilsa2.csd].

### Example 503. Example of the *lposcilsa2* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o lposcilsa2.wav -W ;; for file output any platform
</CsOptions>
```

```
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
Odbfs = 1

instr 1

kcps = 2      ;only integers are allowed
kloop = 0     ;loop start time in samples
kend = 45000  ;loop end time in samples

aenv expsega 0.01, 0.1, 1, 0.1, 0.5, 0.5, 0.01 ;envelope with fast and short segment
aL, aR lposcilsa2 aenv, kcps, kloop, kend, 1 ;use it for amplitude
outs aL, aR

endin
</CsInstruments>
<CsScore>
; Its table size is deferred,
; and format taken from the soundfile header.
f 1 0 0 1 "kickroll.wav" 0 0 0

; Play Instrument #1 for 6 seconds.
; This will loop the drum pattern several times.
i 1 0 2

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*lposcil, lposcila, lposcilsa*

## Credits

Author: Gabriel Maldonado

New in version 5.06

# lpread

`lpread` — Reads a control file of time-ordered information frames.

## Description

Reads a control file of time-ordered information frames.

## Syntax

```
krmsr, krmso, kerr, kcps lpread ktmpnt, ifilcod [, inpoles] [, ifrmrate]
```

## Initialization

*ifilcod* -- integer or character-string denoting a control-file (reflection coefficients and four parameter values) derived from n-pole linear predictive spectral analysis of a source audio signal. An integer denotes the suffix of a file *lp.m*; a character-string (in double quotes) gives a filename, optionally a full pathname. If not fullpath, the file is sought first in the current directory, then in that of the environment variable SADIR (if defined). Memory usage depends on the size of the file, which is held entirely in memory during computation but shared by multiple calls (see also *adsyn*, *pvoc*).

*inpoles* (optional, default=0) -- number of poles in the lpc analysis. It is required only when the control file does not have a header; it is ignored when a header is detected.

*ifrmrate* (optional, default=0) -- frame rate per second in the lpc analysis. It is required only when the control file does not have a header; it is ignored when a header is detected.

## Performance

*lpread* accesses a control file of time-ordered information frames, each containing n-pole filter coefficients derived from linear predictive analysis of a source signal at fixed time intervals (e.g. 1/100 of a second), plus four parameter values:

*krmsr* -- root-mean-square (rms) of the residual of analysis

*krmso* -- rms of the original signal

*kerr* -- the normalized error signal

*kcps* -- pitch in Hz

*ktmpnt* -- The passage of time, in seconds, through the analysis file. *ktmpnt* must always be positive, but can move forwards or backwards in time, be stationary or discontinuous, as a pointer into the analysis file.

*lpread* gets its values from the control file according to the input value *ktmpnt* (in seconds). If *ktmpnt* proceeds at the analysis rate, time-normal synthesis will result; proceeding at a faster, slower, or variable rate will result in time-warped synthesis. At each k-period, *lpread* interpolates between adjacent frames to more accurately determine the parameter values (presented as output) and the filter coefficient settings (passed internally to a subsequent *lpreson*).

The error signal *kerr* (between 0 and 1) derived during predictive analysis reflects the deterministic/random nature of the analyzed source. This will emerge low for pitched (periodic) material and higher for noisy

material. The transition from voiced to unvoiced speech, for example, produces an error signal value of about .3. During synthesis, the error signal value can be used to determine the nature of the *lpreson* driving function: for example, by arbitrating between pitched and non-pitched input, or even by determining a mix of the two. In normal speech resynthesis, the pitched input to *lpreson* is a wideband periodic signal or pulse train derived from a unit such as *buzz*, and the nonpitched source is usually derived from *rand*. However, any audio signal can be used as the driving function, the only assumption of the analysis being that it has a flat response.

*lpfreson* is a formant shifted *lpreson*, in which *kfrqratio* is the (cps) ratio of shifted to original formant positions. This permits synthesis in which the source object changes its apparent acoustic size. *lpfreson* with *kfrqratio* = 1 is equivalent to *lpreson*.

Generally, *lpreson* provides a means whereby the time-varying content and spectral shaping of a composite audio signal can be controlled by the dynamic spectral content of another. There can be any number of *lpread/lpreson* (or *lpfreson*) pairs in an instrument or in an orchestra; they can read from the same or different control files independently.

## Examples

Here is an example of the *lpread* opcode. It uses the file *lpread.csd* [examples/lpread.csd].

### Example 504. Example of the *lpread* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o lpread.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
; works with or without -a option when analyzing "fox.wav" from the manual
ilen filelen "fox.wav" ; length of soundfile
prints "fox.wav = %f seconds\\n",ilen

ktime line 0, p3, p4
krmsr,krms0,kerr,kcps lpread ktime,"fox_poles.lpc"
krms0 = krms0*.00007 ; low volume
aout buzz krms0, kcps, 15, 1
krmsr = krmsr*.0001 ; low volume
asig rand krmsr
outs (aout*2)+asig, (aout*2)+asig ; mix buzz and rand

endin
</CsInstruments>
<CsScore>
; sine
f1 0 4096 10 1
```

```
i 1 0 2.8 1 ; first words only
i 1 4 2.8 2.8 ; whole sentence
e
</CsScore>
</CsoundSynthesizer>
```

The audio file “fox.wav” is 2.8 seconds long. So *filelen*'s output should be a line like this:

```
fox.wav = 2.756667 seconds
```

## See Also

*lpfreson*, *lpreson*, *LPANAL*

# lpreson

lpreson — Resynthesises a signal from the data passed internally by a previous lpread.

## Description

Resynthesises a signal from the data passed internally by a previous *lpread*.

## Syntax

```
ares lpreson asig
```

## Performance

*asig* -- an audio driving function for resynthesis.

*lpreson* receives values internally produced by a leading *lpread*. *lpread* gets its values from the control file according to the input value *ktimpnt* (in seconds). If *ktimpnt* proceeds at the analysis rate, time-normal synthesis will result; proceeding at a faster, slower, or variable rate will result in time-warped synthesis. At each *k*-period, *lpread* interpolates between adjacent frames to more accurately determine the parameter values (presented as output) and the filter coefficient settings (passed internally to a subsequent *lpreson*).

The error signal *kerr* (between 0 and 1) derived during predictive analysis reflects the deterministic/random nature of the analyzed source. This will emerge low for pitched (periodic) material and higher for noisy material. The transition from voiced to unvoiced speech, for example, produces an error signal value of about .3. During synthesis, the error signal value can be used to determine the nature of the *lpreson* driving function: for example, by arbitrating between pitched and non-pitched input, or even by determining a mix of the two. In normal speech resynthesis, the pitched input to *lpreson* is a wideband periodic signal or pulse train derived from a unit such as *buzz*, and the nonpitched source is usually derived from *rand*. However, any audio signal can be used as the driving function, the only assumption of the analysis being that it has a flat response.

*lpfreson* is a formant shifted *lpreson*, in which *kfrqratio* is the (cps) ratio of shifted to original formant positions. This permits synthesis in which the source object changes its apparent acoustic size. *lpfreson* with *kfrqratio* = 1 is equivalent to *lpreson*.

Generally, *lpreson* provides a means whereby the time-varying content and spectral shaping of a composite audio signal can be controlled by the dynamic spectral content of another. There can be any number of *lpread/lpreson* (or *lpfreson*) pairs in an instrument or in an orchestra; they can read from the same or different control files independently.

## Examples

Here is an example of the *lpreson* opcode. It uses the file *lpreson.csd* [examples/lpreson.csd].

### Example 505. Example of the lpreson opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
```

```

<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o lpreson.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
; works with or without -a option when analyzing "fox.wav" from the manual
;both options sound a little different
instr 1

ilen filelen "fox.wav" ; length of soundfile
prints "fox.wav = %f seconds\\n",ilen

ktime line 0, p3, p4
krmsr,krms0,kerr,kcps lpread ktime,"fox_poles.lpc"

krms0 = krms0*.00001 ; low volume
asig buzz krms0, kcps, int(sr/2/kcps), 1 ; max harmonics without aliasing
aout lpreson asig
asig clip aout, 0, 1 ; prevents distortion
outs asig, asig

endin
</CsInstruments>
<CsScore>
; sine
f1 0 4096 10 1

i 1 0 2.8 1 ; first words only
i 1 4 2.8 2.8 ; whole sentence
e
</CsScore>
</CsoundSynthesizer>

```

The audio file “fox.wav” is 2.8 seconds long. So *filelen*'s output should be a line like this:

```
fox.wav = 2.756667 seconds
```

Here is another example of the *lpreson* opcode. It uses the file *lpreson-2.csd* [examples/lpreson-2.csd].

### Example 506. Another example of the *lpreson* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o lpreson-2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100

```

```
kmps = 32
nchnls = 2
Odbfs = 1
; works with or without -a option when analyzing "fox.wav" from the manual
;both options sound a little different
instr 1

ilen filelen "fox.wav" ;length of soundfile 1
prints "fox.wav = %f seconds\\n",ilen

ktime line 0, p3, ilen
krmsr,krms0,kerr,kcps lpread ktime,"fox_nopoles.lpc"
asig disk2 "flute.aiff", 1
aout lpreson asig
ares balance aout, asig
outs ares, ares

endin
</CsInstruments>
<CsScore>

i 1 0 2.8
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*lpfreson, lpread*

# lpshold

lpshold — Generate control signal consisting of held segments.

## Description

Generate control signal consisting of held segments delimited by two or more specified points. The entire envelope is looped at *kfreq* rate. Each parameter can be varied at k-rate.

## Syntax

```
ksig lpshold kfreq, ktrig, iphase, ktime0, kvalue0 [, kvalue1] [, ktime1] [, kvalue2] [, ktime2] [...]
```

## Performance

*k<sub>sig</sub>* -- Output signal

*kfreq* -- Repeat rate in Hz or fraction of Hz

*ktrig* -- If non-zero, retriggers the envelope from start (see *trigger opcode*), before the envelope cycle is completed.

*iphase* -- A value between 0 and 1 to say where to start the loop. Zero, the commonest value, indicates the beginning.

*kvalue0...kvalueN* -- Values of points

*ktime0...ktimeN* -- Times between points; expressed in fractions of a cycle (see below). The final time designates a ramp between the final value and the first value.

*lpshold* is similar to *loopseg*, but can generate only horizontal segments, i.e. holds values for each time interval placed between *ktimeN* and *ktimeN+1*. It can be useful, among other things, for melodic control, like old analog sequencers.

## Examples

Here is an example of the lpshold opcode. It uses the file *lpshold.csd* [examples/lpshold.csd].

### Example 507. Example of the lpshold opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsoundOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o lpshold.wav -W ;; for file output any platform
</CsoundOptions>
<CsoundInstruments>
```

```
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
kfrq init p4 ; frequency of the loop
ifrac = p5 ; fraction of frequency at which to force retrigger
ipan = p6 ; pan position
ktrig metro kfrq * ifrac ; trigger to force restart the loop
iphs = 0 ; initial phase of the loop
; a loop of midi note numbers: note duration etc...
knote lpshold kfrq, ktrig, iphs, 61, 0.0625, 60, 0.9375, 61, 1, 58, 1, \
63, 2, 65, 3
aenv linseg 0,0.01,1,p3-0.11,1,0.1,0 ; amplitude envelope
krnd rspline -0.05,0.05,0.5,1 ; random detune
asig gbuzz 0.2*aenv, cpsmidinn(knote+krnd), 30, 1, 0.5, 1
outs asig*ipan, asig*(1-ipan)
endin

</CsInstruments>
<CsScore>
; cosine wave.
f 1 0 16384 11 1

; 3 layers of the loop are played, each at a different speed,
; - with different retriggering rate, and pan location.
i 1 0 60 0.5 [8/10] 0.5
i 1 0 60 0.375 [8/11] 0.1
i 1 0 60 0.25 [8/13] 0.9
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*loopseg*

## Credits

Author: Gabriel Maldonado

New in Version 4.13

# lpsholdp

lpsholdp — Control signals based on held segments.

## Description

Generate control signal consisting of held segments delimited by two or more specified points. The entire envelope can be looped at time-variant rate. Each segment coordinate can also be varied at k-rate.

## Syntax

```
ksig lpsholdp kphase, kvalue0, ktime0 [, kvalue1] [, ktime1] \  
      [, kvalue2] [, ktime2] [...]
```

## Performance

*ksig* - output signal

*kphase* -- point of the sequence read, expressed as a fraction of a cycle (0 to 1)

*kvalue0...kvalueN* -- Values of points

*ktime0...ktimeN* -- Times between points; expressed in fractions of a cycle (see below). The final time designates a ramp between the final value and the first value.

*lpsholdp* opcode is similar to *lpshold*; the only difference is that, instead of frequency, a time-variant phase is required. If you use a phasor to get the phase value, you will have a behaviour identical to *lpshold*, but interesting results can be achieved when using phases having non-linear motions, making *lpsholdp* more powerful and general than *lpshold*.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# lpslot

*lpslot* — Selects the slot to be use by further lp opcodes.

## Description

Selects the slot to be use by further lp opcodes.

## Syntax

```
lpslot islot
```

## Initialization

*islot* -- number of slot to be selected.

## Performance

*lpslot* selects the slot to be use by further lp opcodes. This is the way to load and reference several analyses at the same time.

## Examples

Here is a typical orc using the opcodes:

```
ipower init 50000 ; Define sound generator
ifreq init 440
asrc buzz ipower, ifreq, 10, 1

vertime line 0, p3, p3 ; Define time lin
lpslot 0 ; Read square data poles
krmsr, krms0, kerr, kcps lpread vertime, "square.pol"
lpslot 1 ; Read triangle data poles
krmsr, krms0, kerr, kcps lpread vertime, "triangle.pol"
kmix line 0, p3, 1 ; Compute result of mixing
lpinterp 0, 1, kmix ; and balance power
ares lpreson asrc
aout balance ares, asrc
out aout
```

## See Also

*lpinterp*

## Credits

Author: Mark Resibois  
Brussels  
1996

New in version 3.44

# lua\_exec

lua\_exec — Executes an arbitrary block of Lua code from the Csound orchestra.



## Note

plugin: Requires lua plugin

## Description

Executes an arbitrary block of Lua code from the Csound orchestra. The code is executed at initialization time, typically from the orchestra header.

## Syntax

```
lua_exec Sluacode
```

## Initialization

*Sluacode* -- A block of Lua code, of any length. Multi-line blocks may be enclosed in double braces (i.e. `{{ }}`). This code is evaluated once at initialization time, typically from the orchestra header. Global and local variables, functions, tables, and classes may be declared and defined. Objects defined at global Lua scope remain in scope throughout the performance, and are visible to any other Lua code in the same Csound thread.

The running instance of Csound is stored as a Lua lightuserdata in a global variable `csound`. This can be passed to any Csound API function. One use of this would be to generate a score using Lua in the orchestra header, and schedule the events for performance using `csoundInputMessage`.



## Note

By default, all objects defined in Lua are defined at global scope. In order to ensure that objects are confined to their own block of code, that is to ensure that the object is visible only in lexical scope, the object must be declared as local. This is the feature of Lua that beginners tend to find the most troublesome.

Another thing to look out for is that Lua arrays use 1-based indexing, not the 0-based indexing used in C and many other programming languages.

## See Also

*lua\_opdef*, *lua\_opcall*.

## Credits

By: Michael Gogins 2011

New in Csound version 5.13.2

# lua\_opdef

lua\_opdef — Define an opcode in Lua at i-rate. The opcode can take any number of output and/or input arguments of any type.

## Description

Define an opcode in Lua at i-rate. The opcode can take any number of output and/or input arguments of any type. The code is executed at initialization time, typically from the orchestra header. Global and local variables, functions, tables, and classes may be declared and defined. Objects defined at global Lua scope remain in scope throughout the performance, and are visible to any other Lua code in the same Csound thread.



### Note

By default, all objects defined in Lua are defined at global scope. In order to ensure that objects are confined to their own block of code, that is to ensure that the object is visible only in lexical scope, the object must be declared as local. This is the feature of Lua that beginners tend to find the most troublesome.

Another thing to look out for is that Lua arrays use 1-based indexing, not the 0-based indexing used in C and many other programming languages.

## Syntax

```
lua_opdef Sname, Sluacode
```

## Initialization

*Sname* -- The name of the opcode.

*Sluacode* -- A block of Lua code, of any length. Multi-line blocks may be enclosed in double braces (i.e. {{ }}). This code is evaluated once at initialization time.

The Lua code must define all functions that will be called from Csound, using the following naming convention, where opcodename stands for the actual opcode name:

- opcodename\_init for the i-rate opcode subroutine.
- opcodename\_kontrol for the k-rate opcode subroutine.
- opcodename\_audio for the a-rate opcode subroutine.
- opcodename\_noteoff for the note-off subroutine.

Each of these Lua functions will receive three lightuserdata (i.e. pointer) arguments: the CSOUND object, the opcode instance, and a pointer to the opcode arguments, which the Lua code must be type cast to a LuaJIT FFI ctype structure containing the opcode output arguments, input arguments, and state variables. Using LuaJIT FFI, the elements of this structure will be accessible as though they were Lua types.

Each of these Lua functions must return 0 for success or 1 for failure.

The Lua functions may do absolutely anything, although of course if real-time performance is expected, care must be taken to disable Lua garbage collection and observe other recommendations for real-time code.

## Example

Here is an example of a Lua opcode, implementing a Moog ladder filter. For purposes of comparison, a user-defined opcode and the native Csound opcode that compute the same sound using the same algorithm also are shown, and timed. The example uses the file *luamoog.csd* [examples/luamoog.csd].

### Example 508. Example of a Lua opcode.

```

<CsoundSynthesizer>
<CsInstruments>
sr = 48000
ksmps = 100
nchnls = 1

gibegan   rtclock

lua_opdef  "moogladder", {{
local ffi = require("ffi")
local math = require("math")
local string = require("string")
local csoundApi = ffi.load('csound64.dll.5.2')
ffi.cdef[[
    int csoundGetKsmps(void *);
    double csoundGetSr(void *);
    struct moogladder_t {
        double *out;
        double *inp;
        double *freq;
        double *res;
        double *istor;
        double sr;
        double ksmps;
        double thermal;
        double f;
        double fc;
        double fc2;
        double fc3;
        double fcr;
        double acr;
        double tune;
        double res4;
        double input;
        double i;
        double j;
        double k;
        double kk;
        double stg[6];
        double delay[6];
        double tanhstg[6];
    };
]]

local moogladder_ct = ffi.typeof('struct moogladder_t *')

function moogladder_init(csound, opcode, carguments)
    local p = ffi.cast(moogladder_ct, carguments)
    p.sr = csoundApi.csoundGetSr(csound)
    p.ksmps = csoundApi.csoundGetKsmps(csound)
    if p.istor[0] == 0 then
        for i = 0, 5 do
            p.delay[i] = 0.0
        end
        for i = 0, 3 do
            p.tanhstg[i] = 0.0
        end
    end
end

```

```

        end
    end
    return 0
end

function moogladder_kontrol(csound, opcode, arguments)
    local p = ffi.cast(moogladder_ct, arguments)
    -- transistor thermal voltage
    p.thermal = 1.0 / 40000.0
    if p.res[0] < 0.0 then
        p.res[0] = 0.0
    end
    -- sr is half the actual filter sampling rate
    p.fc = p.freq[0] / p.sr
    p.f = p.fc / 2.0
    p.fc2 = p.fc * p.fc
    p.fc3 = p.fc2 * p.fc
    -- frequency & amplitude correction
    p.fcr = 1.873 * p.fc3 + 0.4955 * p.fc2 - 0.6490 * p.fc + 0.9988
    p.acr = -3.9364 * p.fc2 + 1.8409 * p.fc + 0.9968
    -- filter tuning
    p.tune = (1.0 - math.exp(-(2.0 * math.pi * p.f * p.fcr))) / p.thermal
    p.res4 = 4.0 * p.res[0] * p.acr
    -- Nested 'for' loops crash, not sure why.
    -- Local loop variables also are problematic.
    -- Lower-level loop constructs don't crash.
    p.i = 0
    while p.i < p.ksmps do
        p.j = 0
        while p.j < 2 do
            p.k = 0
            while p.k < 4 do
                if p.k == 0 then
                    p.input = p.inp[p.i] - p.res4 * p.delay[5]
                    p.stg[p.k] = p.delay[p.k] + p.tune * (math.tanh(p.input * p.thermal) - p.tanhstg[p.k])
                else
                    p.input = p.stg[p.k - 1]
                    p.tanhstg[p.k - 1] = math.tanh(p.input * p.thermal)
                    if p.k < 3 then
                        p.kk = p.tanhstg[p.k]
                    else
                        p.kk = math.tanh(p.delay[p.k] * p.thermal)
                    end
                    p.stg[p.k] = p.delay[p.k] + p.tune * (p.tanhstg[p.k - 1] - p.kk)
                end
                p.delay[p.k] = p.stg[p.k]
                p.k = p.k + 1
            end
            -- 1/2-sample delay for phase compensation
            p.delay[5] = (p.stg[3] + p.delay[4]) * 0.5
            p.delay[4] = p.stg[3]
            p.j = p.j + 1
        end
        p.out[p.i] = p.delay[5]
        p.i = p.i + 1
    end
    return 0
end
}}

/*
Moogladder - An improved implementation of the Moog ladder filter

```

**DESCRIPTION**

This is a new digital implementation of the Moog ladder filter based on the work of Antti Huovilainen, described in the paper "Non-Linear Digital Implementation of the Moog Ladder Filter" (Proceedings of This implementation is probably a more accurate digital representation of the original analogue filter.

## Orchestra Opcodes and Operators

*This is version 2 (revised 14/DEC/04), with improved amplitude/resonance scaling and frequency correction.*

### SYNTAX

ar Moogladder asig, kcf, kres

### PERFORMANCE

asig - input signal  
kcf - cutoff frequency (Hz)  
kres - resonance (0 - 1).

### CREDITS

Victor Lazzarini  
\*/

```

opcode moogladderu, a, akk
asig, kcf, kres  xin
setksmps        1
ipi              = 4 * taninv(1)
/* filter delays */
az1              init 0
az2              init 0
az3              init 0
az4              init 0
az5              init 0
ay4              init 0
amf              init 0
if kres > 1 then
kres              = 1
elseif kres < 0 then
kres              = 0
endif
/* twice the 'thermal voltage of a transistor' */
i2v              = 40000
/* sr is half the actual filter sampling rate */
kfc              = kcf/sr
kf               = kcf/(sr*2)
/* frequency & amplitude correction */
kfcrc            = 1.8730 * (kfc^3) + 0.4955 * (kfc^2) - 0.6490 * kfc + 0.9988
kacrc            = -3.9364 * (kfc^2) + 1.8409 * kfc + 0.9968;
/* filter tuning */
k2vg             = i2v * (1 - exp(-2 * ipi * kfcrc * kf))
/* cascade of 4 1st order sections */
ay1              = az1 + k2vg * (tanh((asig - 4 * kres * amf * kacrc) / i2v) - tanh(az1 / i2v))
az1              = ay1
ay2              = az2 + k2vg * (tanh(ay1 / i2v) - tanh(az2 / i2v))
az2              = ay2
ay3              = az3 + k2vg * (tanh(ay2 / i2v) - tanh(az3 / i2v))
az3              = ay3
ay4              = az4 + k2vg * (tanh(ay3 / i2v) - tanh(az4 / i2v))
az4              = ay4
/* 1/2-sample delay for phase compensation */
amf              = (ay4 + az5) * 0.5
az5              = ay4
/* oversampling */
ay1              = az1 + k2vg * (tanh((asig - 4 * kres * amf * kacrc) / i2v) - tanh(az1 / i2v))
az1              = ay1
ay2              = az2 + k2vg * (tanh(ay1 / i2v) - tanh(az2 / i2v))
az2              = ay2
ay3              = az3 + k2vg * (tanh(ay2 / i2v) - tanh(az3 / i2v))
az3              = ay3
ay4              = az4 + k2vg * (tanh(ay3 / i2v) - tanh(az4 / i2v))
az4              = ay4
amf              = (ay4 + az5) * 0.5
az5              = ay4
xout             amf
endop

```

```

instr 1
    prints "No filter.\n"
    kfe expseg 500, p3*0.9, 1800, p3*0.1, 3000
        kenv linen 10000, 0.05, p3, 0.05
        asig buzz kenv, 100, sr/(200), 1
        ; afil moogladder asig, kfe, 1
        out asig
    endin

instr 2
    prints "Native moogladder.\n"
    kfe expseg 500, p3*0.9, 1800, p3*0.1, 3000
        kenv linen 10000, 0.05, p3, 0.05
        asig buzz kenv, 100, sr/(200), 1
        afil moogladder asig, kfe, 1
        out afil
    endin

instr 3
    prints "UDO moogladder.\n"
    kfe expseg 500, p3*0.9, 1800, p3*0.1, 3000
        kenv linen 10000, 0.05, p3, 0.05
        asig buzz kenv, 100, sr/(200), 1
        afil moogladderu asig, kfe, 1
        out afil
    endin

instr 4
    prints "Lua moogladder.\n"
    kres init 1
    istor init 0
    kfe expseg 500, p3*0.9, 1800, p3*0.1, 3000
        kenv linen 10000, 0.05, p3, 0.05
        asig buzz kenv, 100, sr/(200), 1
        afil init 0
        lua_ikopcall "moogladder", afil, asig, kfe, kres, istor
        out afil
    endin

instr 5
    giended rtclock
    ielapsed = giended - gibegan
    gibegan print ielapsed
    rtclock
    endin

```

</CsInstruments>

<CsScore>

```

f 1 0 65536 10 1
i 5.1 0 1
i 4 1 20
i 5.2 21 1
i 4 22 20
i 5.3 42 1
i 2 43 20
i 5.4 63 1
i 2 64 20
i 5.5 84 1
i 3 85 20
i 5.6 105 1
i 3 106 20
i 5.7 126 1
i 1 127 20
i 5.8 147 1
i 1 148 20
i 5.9 168 1

```

```
i 4      169 20
i 4      170 20
i 4      171 20
e
</CsScore>

</CsoundSynthesizer>
```

## See Also

*lua\_exec, lua\_opcall.*

## Credits

By: Michael Gogins 2011

New in Csound version 5.13.2

## lua\_opcall

lua\_opcall — Calls a Lua opcode at i-rate only. Any number of output and/or input arguments may be passed. All arguments must be passed on the right-hand side. Outputs are returned in the argument.

### Syntax

```
lua_iopcall Sname, ...  
lua_ikopcall Sname, ...  
lua_iaopcall Sname, ...  
lua_iopcall_off Sname, ...  
lua_ikopcall_off Sname, ...  
lua_iaopcall_off Sname, ...
```

### Initialization and Performance

*Sname* -- The name of the opcode.

... -- An arbitrary list of any number of output and input arguments, of any type. The number, sequence, and types of these arguments must agree with the cdef of the arguments structure that is declared in the corresponding lua\_opdef opcode.

lua\_iopcall calls a Lua opcode at i-rate only. Requires `opname_init` to be defined in Lua.

lua\_ikopcall calls a Lua opcode at i-rate and k-rate. Requires `opname_init` and `opname_kontrol` to be defined in Lua.

lua\_iaopcall calls a Lua opcode at i-rate and a-rate. Requires `opname_init` and `opname_audio` to be defined in Lua.

lua\_iopcall\_off calls a Lua opcode at i-rate only. Requires `opname_init` and `opname_noteoff` to be defined in Lua.

lua\_ikopcall\_off calls a Lua opcode at i-rate and k-rate. Requires `opname_init`, `opname_kontrol`, and `opname_noteoff` to be defined in Lua.

lua\_iaopcall\_off calls a Lua opcode at i-rate and a-rate. Requires `opname_init`, `opname_audio`, and `opname_noteoff` to be defined in Lua.

Lua code accesses elements of arguments as follows (pointers to both scalars and arrays are dereferenced by the Lua array access operator):

```
ffi.cdef(' struct arguments_t { double *a_out, double *i_in, double *i_txt, double *f_sig }  
local arguments = ffi.cast("struct arguments_t *", arguments_lightuserdata)  
for i = 0, ksmps -1 do begin arguments.a_out[i] = arguments.i_in[0] * 3 end end
```

The "off" variants of the opcode always schedule a "note off" event that is called when the instrument instance is removed from the active list, and which can be used to release unneeded resources, reschedule the instrument to render a reverb tail, and so on.

## Example

Here is an example of a Lua opcode, showing how to pass strings back and forth between Lua opcodes and Csound orchestra code. The example uses the file *luaopcode.csd* [examples/luaopcode.csd].

### Example 509. Example of a Lua opcode.

```
<CsoundSynthesizer>

<CsInstruments>
lua_opdef "luatest", {{
local ffi = require("ffi")
local string = require("string")
local csoundLibrary = ffi.load('csound64.dll.5.2')
ffi.cdef[[
    int csoundGetKsmpls(void *);
    double csoundGetSr(void *);
    struct luatest_arguments {double *out; double *stringout; char *stringin; double *in1; double *in2;
    ]]
]]
function luatest_init(csound, opcode, carguments)
    local arguments = ffi.cast("struct luatest_arguments *", carguments)
    arguments.sr = csoundLibrary.csoundGetSr(csound)
    print(string.format('stringin: %s', ffi.string(arguments.stringin)))
    print(string.format('sr: %f', arguments.sr))
    arguments.ksmpls = csoundLibrary.csoundGetKsmpls(csound)
    print(string.format('ksmps: %d', arguments.ksmpls))
    arguments.out[0] = arguments.in1[0] * arguments.in2[0]
    ffi.copy(arguments.stringout, 'Hello, world!')
    return 0
end
function luatest_kontrol(csound, opcode, carguments)
    local arguments = ffi.cast("struct luatest_arguments *", carguments)
    arguments.out[0] = arguments.in1[0] * arguments.in2[0]
    return 0
end
function luatest_noteoff(csound, opcode, carguments)
    local arguments = ffi.cast("struct luatest_arguments *", carguments)
    arguments.out[0] = arguments.in1[0] * arguments.in2[0]
    print('off')
    return 0
end
}}

instr 1
    iresult = 0
    Stringin = "stringin"
    Stringout = "stringout"
    lua_iopcall "luatest", iresult, Stringout, Stringin, p2, p3
    prints Stringout
endin
instr 2
    iresult = 0
    Stringin = "stringin"
    Stringout = "initial value"
    lua_iopcall_off "luatest", iresult, Stringout, Stringin, p2, p3
    print iresult
    prints Stringout
endin
</CsInstruments>

<CsScore>
i 1 1 2
i 2 2 2
```

```
i 1 3 2  
e  
</CsScore>  
  
</CsoundSynthesizer>
```

## See Also

*lua\_exec, lua\_opdef.*

## Credits

By: Michael Gogins 2011

New in Csound version 5.13.2

# mac

mac — Multiplies and accumulates a- and k-rate signals.

## Description

Multiplies and accumulates a- and k-rate signals.

## Syntax

```
ares mac ksig1, asig1 [, ksig2] [, asig2] [, ksig3] [, asig3] [...]
```

## Performance

*ksig1, etc.* -- k-rate input signals

*asig1, etc.* -- a-rate input signals

*mac* multiplies and accumulates a- and k-rate signals. It is equivalent to:

$$\text{ares} = \text{asig1} * \text{ksig1} + \text{asig2} * \text{ksig2} + \text{asig3} * \text{ksig3} + \dots$$

## Examples

Here is an example of the mac opcode. It uses the file *mac.csd* [examples/mac.csd]. It is written for \*NIX systems, and will generate errors on Windows.

### Example 510. Example of the mac opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if real audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o mac.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;4 band equalizer

klow = p4 ;low gain1
klmid = p5 ;low gain2
kmidh = p6 ;high gain1
khigh = p7 ;high gain2
```

```

ifn = p8 ;table

ilc1 table 0, ifn ;low frequency range
ilc2 table 1, ifn ;low-mid
ihc1 table 2, ifn ;mid-high
ihc2 table 3, ifn ;high

asig diskin2 "fox.wav", 1
alow1 butterlp asig, ilc1 ;lowpass 1
almid butterlp asig, ilc2 ;lowpass 2
amidh butterhp asig, ihc1 ;highpass 1
ahigh butterhp asig, ihc2 ;highpass 2
aout mac klow, alow1, klmid, almid, kmidh, amidh, khigh, ahigh
      outs aout, aout

endin
</CsInstruments>
<CsScore>
f1 0 4 -2 150 300 600 5000
f2 0 4 -2 75 500 1000 10000
f3 0 4 -2 200 700 1500 3000

;          low lowmid midhigh high table
i 1 0 2.8 2 1 1 1 1
i 1 3 2.8 2 3 1 1 2
i 1 6 2.8 2 1 2 3 3
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*maca*

## Credits

Author: John ffitch  
 University of Bath, Codemist, Ltd.  
 Bath, UK  
 May 1999

New in Csound version 3.54

# maca

maca — Multiply and accumulate a-rate signals only.

## Description

Multiply and accumulate a-rate signals only.

## Syntax

```
ares maca asig1 , asig2 [, asig3] [, asig4] [, asig5] [...]
```

## Performance

*asig1*, *asig2*, ... -- a-rate input signals

*maca* multiplies and accumulates a-rate signals only. It is equivalent to:

$$\text{ares} = \text{asig1} * \text{asig2} + \text{asig3} * \text{asig4} + \text{asig5} * \text{asig6} + \dots$$

## Examples

Here is an example of the maca opcode. It uses the file *maca.csd* [examples/maca.csd]. It is written for \*NIX systems, and will generate errors on Windows.

### Example 511. Example of the maca opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if real audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o mac.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;4 band equalizer

klow = p4 ;low gain1
aenv1  oscil  0.2, 1, 4
aenv1 = aenv1 + klow
klmid = p5 ;low gain2
aenv2  oscil  0.21, 1.1, 4
aenv2 = aenv2 + klmid
```

```

kmidh = p6 ;high gain1
aenv3  oscil  0.19, 1.2, 4
aenv3 = aenv3 + kmidh
khigh = p7 ;high gain2
aenv4  oscil  0.18, 1.3, 4
aenv4 = aenv4 + khigh

ifn = p8 ;table

ilc1 table 0, ifn ;low frequency range
ilc2 table 1, ifn ;low-mid
ihc1 table 2, ifn ;mid-high
ihc2 table 3, ifn ;high

asig diskin2 "fox.wav", 1
alow1 butterlp asig, ilc1 ;lowpass 1
almid butterlp asig, ilc2 ;lowpass 2
amidh butterhp asig, ihc1 ;highpass 1
ahigh butterhp asig, ihc2 ;highpass 2
aout maca aenv1, alow1, aenv2, almid, aenv3, amidh, aenv4, ahigh
      outs aout, aout

endin
</CsInstruments>
<CsScore>
f1 0 4 -2 150 300 600 5000
f2 0 4 -2 75 500 1000 10000
f3 0 4 -2 200 700 1500 3000
f4 0 4096 10 1

;          low lowmid midhigh high table
i 1 0 2.8 2 1 1 1 1
i 1 3 2.8 2 3 1 1 2
i 1 6 2.8 2 1 2 3 3
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*mac*

## Credits

Author: John ffitch  
 University of Bath, Codemist, Ltd.  
 Bath, UK  
 May 1999

New in Csound version 3.54

## madsr

`madsr` — Calculates the classical ADSR envelope using the *linsegr* mechanism.

## Description

Calculates the classical ADSR envelope using the *linsegr* mechanism.

## Syntax

```
ares madsr iatt, idec, islev, irel [, idel] [, ireltim]
```

```
kres madsr iatt, idec, islev, irel [, idel] [, ireltim]
```

## Initialization

*iatt* -- duration of attack phase

*idec* -- duration of decay

*islev* -- level for sustain phase

*irel* -- duration of release phase.

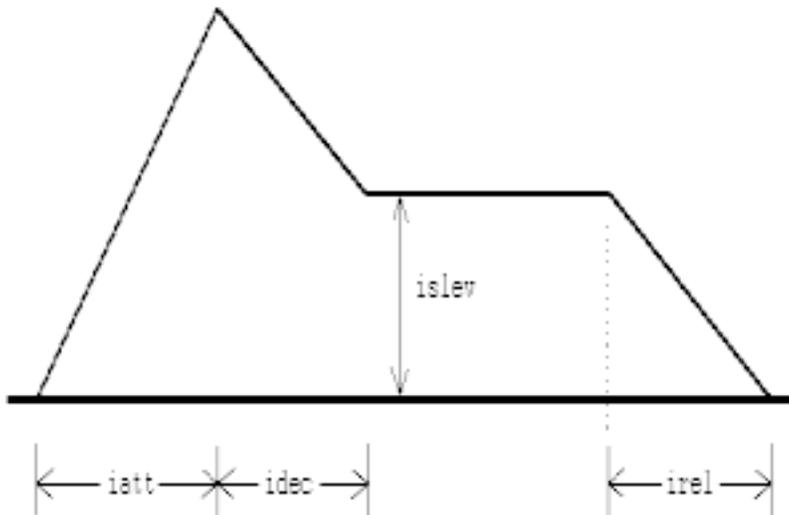
*idel* -- period of zero before the envelope starts

*ireltim* (optional, default=-1) -- Control release time after receiving a MIDI noteoff event. If less than zero, the longest release time given in the current instrument is used. If zero or more, the given value will be used for release time. Its default value is -1. (New in Csound 3.59 - not yet properly tested)

Please note that the release time cannot be longer than  $32767/kr$  seconds.

## Performance

The envelope is in the range 0 to 1 and may need to be scaled further. The envelope may be described as:



Picture of an ADSR envelope.

The length of the sustain is calculated from the length of the note. This means *adsr* is not suitable for use with MIDI events. The opcode *madsr* uses the *linsegr* mechanism, and so can be used in MIDI applications.

You can use other pre-made envelopes which start a release segment upon receiving a note off message, like *linsegr* and *expsegr*, or you can construct more complex envelopes using *xtratim* and *release*. Note that you don't need to use *xtratim* if you are using *madsr*, since the time is extended automatically.



## Note

Times for *iatt*, *idec* and *irel* cannot be 0. If 0 is used, no envelope is generated. Use a very small value like 0.0001 if you need an instantaneous attack, decay or release.

## Examples

Here is an example of the *madsr* opcode. It uses the file *madsr.csd* [examples/madsr.csd].

### Example 512. Example of the *madsr* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o madsr.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

/* Written by Iain McCurdy */
; Initialize the global variables.
sr = 44100
kr = 441
ksmps = 100
nchnls = 1

; Instrument #1.
instr 1
; Attack time.
iattack = 0.5
; Decay time.
idecay = 0
; Sustain level.
isustain = 1
; Release time.
irelease = 0.5
aenv madsr iattack, idecay, isustain, irelease

a1 oscili 10000, 440, 1
out a1*aenv
endin

</CsInstruments>
<CsScore>

/* Written by Iain McCurdy */
```

```
; Table #1, a sine wave.
f 1 0 1024 10 1

; Leave the score running for 6 seconds.
f 0 6

; Play Instrument #1 for two seconds.
i 1 0 2
e

</CsScore>
</CsoundSynthesizer>
```

Here is another example of the `madsr` opcode, using midi input. It uses the file `madsr-2.csd` [examples/madsr-2.csd].

### Example 513. second example of the `madsr` opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0   ;;realtime audio out and realtime midi in
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o madsr-2.wav -W   ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

icps cpsmidi
iamp ampmidi .5

kenv madsr 0.5, 0, 1, 0.5
asig pluck kenv, icps, icps, 2, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 2 0 4096 10 1

f0 30 ;runs 30 seconds
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*linsegr, expsegr, envlpxr, mxadsr, madsr, xadsr expon, expseg, expsega line, linseg, xtratim*

## Credits

Author: John ffitich

November 2002. Thanks to Rasmus Ekman, added documentation for the *ireltim* parameter.

December 2002. Thanks to Iain McCurdy, added an example.

December 2002. Thanks to Istvan Varga, added documentation about the maximum release time.

New in Csound version 3.49.

# mags

mags — Obtains the magnitudes of a complex-number array

## Description

This opcode returns the magnitudes of a complex-number array (in rfft format), as a real-valued array with half the size of its input plus one. The magnitude for the Nyquist frequency are kept in the last position of the array.

## Syntax

```
kout[] mags kin[]
```

## Performance

*kout[]* -- output array containing the magnitudes (size = input\_size/2 + 1). It will be created if it does not exist.

*kin[]* -- input array containing the complex-valued real-imaginary input.

## Examples

Here is an example of the mags opcode. It uses the file *mags.csd* [examples/mags.csd].

### Example 514. Example of the mags opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-d -o dac
</CsOptions>
<CsInstruments>
/* ksmpls needs to be an integer div of
   hopsize */
ksmpls = 64

instr 1

   ihopsize = 256 ; hopsize
   ifftsize = 2048 ; FFT size
   iolaps = ifftsize/ihopsize ; overlaps
   ibw = sr/ifftsize ; bin bandwidth
   kcnt init 0 ; counting vars
   krow init 0

   kOla[] init ifftsize ; overlap-add buffer
   kIn[] init ifftsize ; input buffer
   kOlph[] init ifftsize/2+1 ; old phases
   kPhs[] init ifftsize/2+1 ; synthesis phases
   kDeltaOut[] init ifftsize/2+1 ; synthesis freqs
   kMagsOut[] init ifftsize/2+1 ; synthesis mags
   kOut[][] init iolaps, ifftsize ; output buffers

   a1 diskin2 "fox.wav",1,0,1 ; audio input
```

```

/* every hopsize samples */
if kcnt == ihopsize then
  /* window and take FFT */
  kWin[] window kIn,krow*ihopsize
  kSpec[] rfft kWin

  /* mags & phases */
  kMags[] mags kSpec
  kPha[] phs kSpec

  /* phase diffs */
  kDelta[] = kPha - kOlph
  kOlph = kPha

  /* pitch shift */
  ki = 0
  iscal = 1.25 ; maj 3rd up
  until ki == ifftsize/2 do
    if ki*iscal < ifftsize/2 then
      kDeltaOut[ki*iscal] = kDelta[ki]*iscal
      kMagsOut[ki*iscal] = kMags[ki]
    endif
    ki += 1
  od

  /* accum phases */
  kPhs = kDeltaOut + kPhs
  kSpec pol2rect kMagsOut,kPhs

  /* IFFT + window */
  kRow[] rfft kSpec
  kWin window kRow, krow*ihopsize
  /* place it on out buffer */
  kOut setrow kWin, krow

  /* zero the ola buffer */
  kOla = 0
  /* overlap-add */
  ki = 0
  until ki == iolaps do
    kRow getrow kOut, ki
    kOla = kOla + kRow
    ki += 1
  od

  /* update counters */
  krow = (krow+1)%iolaps
  kcnt = 0
endif

/* shift audio in/out of buffers */
kIn shiftin a1
a2 shiftout kOla
a1 delay a1, (ifftsize+ihopsize)/sr
out a1*0.3+a2/iolaps

/* increment counter */
kcnt += ksmpps

endin

</CsInstruments>
<CsScore>
i1 0 10
</CsScore>
</CsoundSynthesizer>

```

## See Also

Vectorial opcodes, array opcodes

## Credits

Author: Victor Lazzarini  
NUI Maynooth  
2014

New in version 6.04

# mandel

mandel — Mandelbrot set

## Description

Returns the number of iterations corresponding to a given point of complex plane by applying the Mandelbrot set formula.

## Syntax

```
kiter, koutrig mandel ktrig, kx, ky, kmaxIter
```

## Performance

*kiter* - number of iterations

*koutrig* - output trigger signal

*ktrig* - input trigger signal

*kx, ky* - coordinates of a given point belonging to the complex plane

*kmaxIter* - maximum iterations allowed

*mandel* is an opcode that allows the use of the Mandelbrot set formula to generate an output that can be applied to any musical (or non-musical) parameter. It has two output arguments: *kiter*, that contains the iteration number of a given point, and *koutrig*, that generates a trigger 'bang' each time *kiter* changes. A new number of iterations is evaluated only when *ktrig* is set to a non-zero value. The coordinates of the complex plane are set in *kx* and *ky*, while *kmaxIter* contains the maximum number of iterations. Output values, which are integer numbers, can be mapped in any sorts of ways by the composer.

## Examples

Here is an example of the mandel opcode. It uses the file *mandel.csd* [examples/mandel.csd].

### Example 515. Example of the mandel opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o mandel.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
;example by Brian Evans
sr = 44100
ksmps = 32
nchnls = 2

instr 1
```

```

ipitchtable = 1      ; pitch table in score
ipitchndx = p5      ; p5=pitch index from table

ipitch table ipitchndx, ipitchtable
kenv expseg 1.0, 1.0, 1.0, 11.5, .0001
asig pluck ampdb(p4)*kenv, cpspch(ipitch), cpspch(ipitch), 0, 1
      outs asig, asig

endin
</CsInstruments>
<CsScore>

f1 0 32 -2 6.00 6.02 6.04 6.05 6.07 6.09 6.11 ; f1 is a pitch table defining a four octave C major sca
      7.00 7.02 7.04 7.05 7.07 7.09 7.11 ; on C two octaves below middle C
      8.00 8.02 8.04 8.05 8.07 8.09 8.11
      9.00 9.02 9.04 9.05 9.07 9.09 9.11

;ins start dur ampdb(p4) pitchndx(p5)

i1 0 12.0 75 3
i1 1.5999 12.0 75 4
i1 3.4000 12.0 75 5
i1 4.2000 12.0 75 6
i1 4.4000 12.0 75 7
i1 4.6000 12.0 75 9
i1 4.8000 12.0 75 10
i1 5.0000 12.0 75 5
i1 5.2000 12.0 75 27
i1 5.4000 12.0 75 5
i1 5.6000 12.0 75 20
i1 6.0000 12.0 75 24
i1 6.2000 12.0 75 2
i1 6.4000 12.0 75 27
i1 6.6000 12.0 75 20
i1 6.8000 12.0 75 15
i1 7.0000 12.0 75 3
i1 7.2000 12.0 75 3
i1 7.4000 12.0 75 23
i1 7.6000 12.0 75 9
i1 7.8000 12.0 75 17
i1 8.0000 12.0 75 18
i1 8.2000 12.0 75 3
i1 8.4000 12.0 75 26
i1 8.6000 12.0 75 15
i1 8.8 12.0 75 2
i1 9 12.0 75 26
i1 9.2 12.0 75 8
i1 9.3999 12.0 75 22
i1 9.5999 12.0 75 22
i1 9.7999 12.0 75 20
i1 9.9999 12.0 75 19
i1 10.399 12.0 75 20
i1 10.799 12.0 75 22
i1 10.999 12.0 75 27
i1 11.199 12.0 75 25
i1 11.399 12.0 75 20
i1 11.599 12.0 75 21
i1 11.799 12.0 75 24
i1 11.999 12.0 75 24
i1 12.199 12.0 75 4
i1 12.399 12.0 75 13
i1 12.599 12.0 75 15
i1 12.799 12.0 75 14
i1 12.999 12.0 75 3
i1 13.199 12.0 75 21
i1 13.399 12.0 75 6

```

```
i1 13.599 12.0 75 3
i1 13.799 12.0 75 10
i1 13.999 12.0 75 25
i1 14.199 12.0 75 21
i1 14.399 12.0 75 20
i1 14.599 12.0 75 19
i1 14.799 12.0 75 18
i1 15.199 12.0 75 17
i1 15.599 12.0 75 16
i1 15.999 12.0 75 15
i1 16.599 12.0 75 14
i1 17.199 12.0 75 13
i1 18.399 12.0 75 12
i1 18.599 12.0 75 11
i1 19.199 12.0 75 10
i1 19.799 12.0 75 9
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

More information on this opcode: *Composing Fractal Music with Csound* [<http://mymbs.mbs.net/~pfisher/FOV2-0010016C/FOV2-0010016E/FOV2-001001A3/chapters/35evans/intro.html>], by Brian Evans

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# mandol

mandol — An emulation of a mandolin.

## Description

An emulation of a mandolin.

## Syntax

```
ares mandol kamp, kfreq, kpluck, kdetune, kgain, ksize \  
    [, ifn] [, iminfreq]
```

## Initialization

*ifn* -- table number containing the pluck wave form. The file *mandpluk.aiff* [examples/mandpluk.aiff] is suitable for this. It is also available at <ftp://ftp.cs.bath.ac.uk/pub/dream/documentation/sounds/modelling/>.

*iminfreq* (optional, default=0) -- Lowest frequency to be played on the note. If it is omitted it is taken to be the same as the initial *kfreq*.

## Performance

*kamp* -- Amplitude of note.

*kfreq* -- Frequency of note played.

*kpluck* -- The pluck position, in range 0 to 1. Suggest 0.4.

*kdetune* -- The proportional detuning between the two strings. Suggested range 0.9 to 1.

*kgain* -- the loop gain of the model, in the range 0.97 to 1.

*ksize* -- The size of the body of the mandolin. Range 0 to 2.

## Examples

Here is an example of the mandol opcode. It uses the file *mandol.csd* [examples/mandol.csd], and *mandpluk.aiff* [examples/mandpluk.aiff].

### Example 516. Example of the mandol opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
-odac      ;;realtime audio out  
;-iadc     ;;uncomment -iadc if realtime audio input is needed too  
; For Non-realtime ouput leave only the line below:  
; -o mandol.wav -W ;; for file output any platform
```

```
</CsOptions>
<CsInstruments>

  sr = 44100
  ksmps = 32
  nchnls = 2
  0dbfs = 1

  instr 1

  kamp = p4
  ksize = p5
  kdetune = p6
  asig mandol kamp, 880, .4, kdetune, 0.99, ksize, 1, 220
    outs asig, asig

  endin
</CsInstruments>
<CsScore>
  ; "mandpluk.aiff" audio file
  f 1 0 8192 1 "mandpluk.aiff" 0 0 0

  i 1 .5 1 1 2 .99
  i 1 + 1 .5 1 .99 ;lower volume to compensate
  i 1 + 3 .3 .3 .99 ;lower volume to compensate

  i 1 4 1 1 2 .39 ;change detune value
  i 1 + 1 .5 1 .39
  i 1 + 3 .3 .3 .39
  e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitich (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

# maparray

maparray — Apply a function to every element of a vector.

## Description

Apply a function of one argument to every element of a vector (one-dimensional k-rate array).

## Syntax

```
karray maparray kinarray, String  
karray maparray_i kinarray, String
```

## Initialization

*String* -- a string that names an opcode function, at i-rate for maparray\_i or k-rate for maparray.

## Performance

*karray* -- array for answers.

*kinarray* -- array for arguments to the function.

## Examples

Here is an example of the maparray opcode. It uses the file *maparray.csd* [examples/maparray.csd].

### Example 517. Example of the maparray opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
-n  
</CsOptions>  
<CsInstruments>  
  
sr = 44100  
ksmps = 32  
nchnls = 2  
0dbfs = 1  
  
instr 1  
  
;create an array and fill with numbers  
kArrSrc[] fillarray 1.01, 2.02, 3.03, 4.05, 5.08, 6.13, 7.21  
  
;print source array  
printf "%s", 1, "\nSource array:\n"  
kndx = 0  
until kndx == lenarray(kArrSrc) do  
printf "kArrSrc[%d] = %f\n", kndx+1, kndx, kArrSrc[kndx]
```

```

kndx   +=      1
      od

;create an empty array for the results
kArrRes[] init 7

;apply the sqrt() function to each element
kArrRes maparray kArrSrc, "sqrt"

;print the result
      printf "%s", 1, "\nResult after applying sqrt() to source array\n"
kndx   =      0
      until kndx == lenarray(kArrRes) do
      printf "kArrRes[%d] = %f\n", kndx+1, kndx, kArrRes[kndx]
kndx   +=      1
      od

;apply the log() function to each element
kArrRes maparray kArrSrc, "log"

;print the result
      printf "%s", 1, "\nResult after applying log() to source array\n"
kndx   =      0
      until kndx == lenarray(kArrRes) do
      printf "kArrRes[%d] = %f\n", kndx+1, kndx, kArrRes[kndx]
kndx   +=      1
      od

;apply the int() function to each element
kArrRes maparray kArrSrc, "int"

;print the result
      printf "%s", 1, "\nResult after applying int() to source array\n"
kndx   =      0
      until kndx == lenarray(kArrRes) do
      printf "kArrRes[%d] = %f\n", kndx+1, kndx, kArrRes[kndx]
kndx   +=      1
      od

;apply the frac() function to each element
kArrRes maparray kArrSrc, "frac"

;print the result
      printf "%s", 1, "\nResult after applying frac() to source array\n"
kndx   =      0
      until kndx == lenarray(kArrRes) do
      printf "kArrRes[%d] = %f\n", kndx+1, kndx, kArrRes[kndx]
kndx += 1
      od

;turn instrument instance off
      turnoff

endin

</CsInstruments>
<CsScore>
i 1 0 0.1
e
</CsScore>
</CsSoundSynthesizer>

```

## See Also

Vectorial opcodes

## Credits

Author: John ffitch  
Codemist Ltd  
2013

New in version 6.00

# marimba

marimba — Physical model related to the striking of a wooden block.

## Description

Audio output is a tone related to the striking of a wooden block as found in a marimba. The method is a physical model developed from Perry Cook but re-coded for Csound.

## Syntax

```
ares marimba kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivibfn, idec \  
    [, idoubles] [, itriples]
```

## Initialization

*ihrd* -- the hardness of the stick used in the strike. A range of 0 to 1 is used. 0.5 is a suitable value.

*ipos* -- where the block is hit, in the range 0 to 1.

*imp* -- a table of the strike impulses. The file *marmstk1.wav* [examples/marmstk1.wav] is a suitable function from measurements and can be loaded with a *GENOI* table. It is also available at ftp://ftp.c-s.bath.ac.uk/pub/dream/documentation/sounds/modelling/.

*ivfn* -- shape of vibrato, usually a sine table, created by a function

*idec* -- time before end of note when damping is introduced

*idoubles* (optional) -- percentage of double strikes. Default is 40%.

*itriples* (optional) -- percentage of triple strikes. Default is 20%.

## Performance

*kamp* -- Amplitude of note.

*kfreq* -- Frequency of note played.

*kvibf* -- frequency of vibrato in Hertz. Suggested range is 0 to 12

*kvamp* -- amplitude of the vibrato

## Examples

Here is an example of the marimba opcode. It uses the file *marimba.csd* [examples/marimba.csd], and *marmstk1.wav* [examples/marmstk1.wav].

### Example 518. Example of the marimba opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform
```

```

; Audio out   Audio in   No messages
-odac         -iadc      -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o marimba.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
ksmps = 128
nchnls = 2

; Instrument #1.
instr 1
  ifreq = cpspch(p4)
  ihrd = 0.1
  ipos = 0.561
  imp = 1
  kvibf = 6.0
  kvamp = 0.05
  ivibfn = 2
  idec = 0.6

  a1 marimba 20000, ifreq, ihrd, ipos, imp, kvibf, kvamp, ivibfn, idec, 20, 10

  outs a1, a1
endin

</CsInstruments>
<CsScore>

; Table #1, the "marmstk1.wav" audio file.
f 1 0 256 1 "marmstk1.wav" 0 0 0
; Table #2, a sine wave for the vibrato.
f 2 0 128 10 1

; Play Instrument #1 for one second.
i 1 0 1 8.09
i 1 + 0.5 8.00
i 1 + 0.5 7.00
i 1 + 0.25 8.02
i 1 + 0.25 8.01
i 1 + 0.25 7.09
i 1 + 0.25 8.02
i 1 + 0.25 8.01
i 1 + 0.25 7.09
i 1 + 0.3333 8.09
i 1 + 0.3333 8.02
i 1 + 0.3334 8.01
i 1 + 0.25 8.00
i 1 + 0.3333 8.09
i 1 + 0.3333 8.02
i 1 + 0.25 8.01
i 1 + 0.3333 7.00
i 1 + 0.3334 6.00

e

</CsScore>
</CsSoundSynthesizer>

```

## See Also

*vibes*

## Credits

Author: John fitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

# massign

`massign` — Assigns a MIDI channel number to a Csound instrument.

## Description

Assigns a MIDI channel number to a Csound instrument.

## Syntax

```
massign ichnl, insnum[, ireset]  
massign ichnl, "insname"[, ireset]
```

## Initialization

*ichnl* -- MIDI channel number (1-16).

*insnum* -- Csound orchestra instrument number. If zero or negative, the channel is muted (i.e. it doesn't trigger a csound instrument, though information will still be received by opcodes like *midin*).

*"insname"* -- A string (in double-quotes) representing a named instrument.

*ireset* -- If non-zero resets the controllers; default is to reset.

## Performance

Assigns a MIDI channel number to a Csound instrument. Also useful to make sure a certain instrument (if its number is from 1 to 16) will not be triggered by midi noteon messages (if using something *midin* to interpret midi information). In this case set *insnum* to 0 or a negative number.

If *ichan* is set to 0, the value of *insnum* is used for all channels. This way you can route all MIDI channels to a single Csound instrument. You can also disable triggering of instruments from MIDI note events from all channels with the following line:

```
massign 0,0
```

This can be useful if you are doing all MIDI evaluation within Csound with an always on instrument(e.g. using *midin* and *turnon*) to avoid doubling the instrument when a note is played.

## Examples

Here is an example of the `massign` opcode. It uses the file *massign.csd* [examples/massign.csd].

### Example 519. Example of the `massign` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
```

```

<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0    ;;realtime audio out and realtime midi in
;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
;-o massign.wav -W    ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giengine fluidEngine
; soundfont path to manual/examples
isfnum fluidLoad "19Trumpet.sf2", giengine, 1
fluidProgramSelect giengine, 1, isfnum, 0, 56

massign 0,0 ;disable triggering of all instruments on all channels, but
massign 12,10 ;assign instr. 10 to midi channel 12
massign 3,30 ;assign instr. 30 to midi channel 3

instr 10 ; soundfont only on midi channel 12

mididefault 60, p3
midinoteonkey p4, p5 ; in midi notes
ikey init p4
ivel init p5
fluidNote giengine, 1, ikey, ivel
endin

instr 30 ; FM-oscillator only on midi channel 3

mididefault 60, p3
midinoteoncps p4, p5 ; in Hertz
icps init p4
iamp init p5
iamp = iamp/127
kenv madsr 0.5, 0, 1, 0.5
asig foscil iamp*kenv, icps, 1, 1.414, 2, 1
outs asig, asig
endin

instr 99 ; output sound from fluidengine

imvol init 7
aL, aR fluidOut giengine
outs aL*imvol, aR*imvol
endin
</CsInstruments>
<CsScore>
; sine
f 1 0 16384 10 1

i 10 0 2 60 100 ;one note on the trumpet in midi and...
i 30 2 2 220 80 ;one FM note in Hz
i 99 0 60 ;stay active for 60 sec.
e

</CsScore>
</CsSoundSynthesizer>

```

## See Also

*ctrlinit*

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT, Cambridge, Mass.

New in Csound version 3.47

*ireset* parameter new in Csound5

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel range.

# max

max — Produces a signal that is the maximum of any number of input signals.

## Description

The *max* opcode takes any number of a-rate, k-rate or i-rate signals as input (all of the same rate), and outputs a signal at the same rate that is the maximum of all of the inputs. For a-rate signals, the inputs are compared one sample at a time (i.e. *max* does not scan an entire ksmps period of a signal for its local maximum as the *max\_k* opcode does).

## Syntax

```
amax max ain1, ain2 [, ain3] [, ain4] [...]
```

```
kmax max kin1, kin2 [, kin3] [, kin4] [...]
```

```
imax max iin1, iin2 [, iin3] [, iin4] [...]
```

## Performance

*ain1, ain2, ...* -- a-rate signals to be compared.

*kin1, kin2, ...* -- k-rate signals to be compared.

*iin1, iin2, ...* -- i-rate signals to be compared.

## Examples

Here is an example of the max opcode. It uses the file *max.csd* [examples/max.csd].

### Example 520. Example of the max opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o max.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

k1  oscili 1, 10.0, 1   ;combine 3 sinusses
k2  oscili 1, 1.0, 1   ;at different rates
```

```
k3  oscili 1, 3.0, 1
kmax max k1, k2, k3
kmax = kmax*250 ;scale kmax
printk2 kmax ;check the values

aout vco2 .5, 220, 6 ;sawtooth
asig moogvcf2 aout, 600+kmax, .5 ;change filter around 600 Hz
outs asig, asig

endin

</CsInstruments>
<CsScore>
f1 0 32768 10 1

i1 0 5
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*min, maxabs, minabs, maxaccum, minaccum, maxabsaccum, minabsaccum, max\_k*

## Credits

Author: Anthony Kozar  
March 2006

New in Csound version 5.01; i-rate version new in 6.04

# maxabs

maxabs — Produces a signal that is the maximum of the absolute values of any number of input signals.

## Description

The *maxabs* opcode takes any number of a-rate or k-rate signals as input (all of the same rate), and outputs a signal at the same rate that is the maximum of all of the inputs. It is identical to the *max* opcode except that it takes the absolute value of each input before comparing them. Therefore, the output is always non-negative. For a-rate signals, the inputs are compared one sample at a time (i.e. *maxabs* does not scan an entire ksmps period of a signal for its local maximum as the *max\_k* opcode does).

## Syntax

```
amax maxabs ain1, ain2 [, ain3] [, ain4] [...]  
kmax maxabs kin1, kin2 [, kin3] [, kin4] [...]
```

## Performance

*ain1, ain2, ...* -- a-rate signals to be compared.

*kin1, kin2, ...* -- k-rate signals to be compared.

## Examples

Here is an example of the maxabs opcode. It uses the file *maxabs.csd* [examples/maxabs.csd].

### Example 521. Example of the maxabs opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
-odac   ;;realtime audio out  
;-iadc   ;;uncomment -iadc if realtime audio input is needed too  
; For Non-realtime ouput leave only the line below:  
; -o maxabs.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>  
  
sr = 44100  
ksmps = 32  
nchnls = 2  
0dbfs = 1  
  
instr 1  
  
k1  oscili 1, 10.0, 1   ;combine 3 sinusses  
k2  oscili 1, 1.0, 1   ;at different rates  
k3  oscili 1, 3.0, 1  
kmax maxabs k1, k2, k3  
kmax = kmax*250      ;scale kmax
```

```
printk2 kmax      ;check the values

aout vco2 .5, 220, 6 ;sawtooth
asig moogvcf2 aout, 600+kmax, .5 ;change filter above 600 Hz
outs asig, asig

endin

</CsInstruments>
<CsScore>
f1 0 32768 10 1

i1 0 5
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*minabs, max, min, maxaccum, minaccum, maxabsaccum, minabsaccum, max\_k*

## Credits

Author: Anthony Kozar  
March 2006

New in Csound version 5.01

# maxabsaccum

maxabsaccum — Accumulates the maximum of the absolute values of audio signals.

## Description

*maxabsaccum* compares two audio-rate variables and stores the maximum of their absolute values into the first.

## Syntax

```
maxabsaccum aAccumulator, aInput
```

## Performance

*aAccumulator* -- audio variable to store the maximum value

*aInput* -- signal that *aAccumulator* is compared to

The *maxabsaccum* opcode is designed to accumulate the maximum value from among many audio signals that may be in different note instances, different channels, or otherwise cannot all be compared at once using the *maxabs* opcode. *maxabsaccum* is identical to *maxaccum* except that it takes the absolute value of *aInput* before the comparison. Its semantics are similar to *vincr* since *aAccumulator* is used as both an input and an output variable, except that *maxabsaccum* keeps the maximum absolute value instead of adding the signals together. *maxabsaccum* performs the following operation on each pair of samples:

```
if (abs(aInput) > aAccumulator) aAccumulator = abs(aInput)
```

*aAccumulator* will usually be a global audio variable. At the end of any given computation cycle (*k*-period), after its value is read and used in some way, the accumulator variable should usually be reset to zero (perhaps by using the *clear* opcode). Clearing to zero is sufficient for *maxabsaccum*, unlike the *maxaccum* opcode.

## Examples

Here is an example of the *maxabsaccum* opcode. It uses the file *maxabsaccum.csd* [examples/maxabsaccum.csd].

### Example 522. Example of the maxabsaccum opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o maxabsaccum.wav -W ;; for file output any platform
```

```
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;saw

asig vco2 .2, p4
outs asig, asig
gasaw = asig
endin

instr 2 ;sine

aout poscil .3, p4, 1
outs aout, aout
gasin = aout
endin

instr 10

accum init 0
maxabsaccum accum, gasaw + gasin ;saw and sine accumulated
accum dcblock2 accum ;get rid of DC
outs accum, accum

clear accum
endin

</CsInstruments>
<CsScore>
f 1 0 4096 10 1 ;sine wave

i 1 0 7 330
i 2 3 3 440

i 1 10 7 330 ;same notes but without maxabsaccum, for comparison
i 2 13 3 440

i 10 0 6 ;accumulation note stops after 6 seconds

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*minabsaccum, maxaccum, minaccum, max, min, maxabs, minabs, vincr, clear*

## Credits

Author: Anthony Kozar  
March 2006

New in Csound version 5.01

# maxaccum

maxaccum — Accumulates the maximum value of audio signals.

## Description

*maxaccum* compares two audio-rate variables and stores the maximum value between them into the first.

## Syntax

```
maxaccum aAccumulator, aInput
```

## Performance

*aAccumulator* -- audio variable to store the maximum value

*aInput* -- signal that *aAccumulator* is compared to

The *maxaccum* opcode is designed to accumulate the maximum value from among many audio signals that may be in different note instances, different channels, or otherwise cannot all be compared at once using the *max* opcode. Its semantics are similar to *vincr* since *aAccumulator* is used as both an input and an output variable, except that *maxaccum* keeps the maximum value instead of adding the signals together. *maxaccum* performs the following operation on each pair of samples:

```
if (aInput > aAccumulator) aAccumulator = aInput
```

*aAccumulator* will usually be a global audio variable. At the end of any given computation cycle (k-period), after its value is read and used in some way, the accumulator variable should usually be reset to zero (perhaps by using the *clear* opcode). Care must be taken however if *aInput* is negative at any point, in which case the accumulator should be initialized and reset to some large enough negative value that will always be less than the input signals to which it is compared.

## Examples

Here is an example of the *maxaccum* opcode. It uses the file *maxaccum.csd* [examples/maxaccum.csd].

### Example 523. Example of the *maxabs* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o maxaccum.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;saw

asig vco2 .2, p4
outs asig, asig
gasaw = asig
endin

instr 2 ;sine

aout poscil .3, p4, 1
outs aout, aout
gasin = aout
endin

instr 10

accum init 0
maxaccum accum, gasaw + gasin ;saw and sine accumulated
outs accum, accum

clear accum
endin

</CsInstruments>
<CsScore>
f 1 0 4096 10 1

i 1 0 7 330
i 2 3 3 440

i 1 10 7 330 ;same notes but without maxaccum, for comparison
i 2 13 3 440

i 10 0 6 ;accumulation note stops after 6 seconds

e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*minaccum, maxabsaccum, minabsaccum, max, min, maxabs, minabs, vincer, clear*

## Credits

Author: Anthony Kozar  
March 2006

New in Csound version 5.01

# maxalloc

maxalloc — Limits the number of allocations of an instrument.

## Description

Limits the number of allocations of an instrument.

## Syntax

```
maxalloc insnum, icount  
maxalloc Sinsname, icount
```

## Initialization

```
insnum -- instrument number  
Sinsname -- instrument name  
icount -- number of instrument allocations
```

## Performance

*maxalloc* limits the number of simultaneous instances (notes) of an instrument. Any score events after the maximum has been reached, are ignored.

All instances of *maxalloc* must be defined in the header section, not in the instrument body.

## Examples

Here is an example of the maxalloc opcode. It uses the file *maxalloc.csd* [examples/maxalloc.csd].

### Example 524. Example of the maxalloc opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
-odac      ;;realtime audio out  
;-iadc     ;;uncomment -iadc if realtime audio input is needed too  
; For Non-realtime ouput leave only the line below:  
; -o maxalloc.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>  
  
sr = 44100  
ksmps = 32  
nchnls = 2  
0dbfs = 1  
  
maxalloc 1, 3 ; Limit to three instances.
```

```
instr 1
  asig oscil .3, p4, 1
      outs asig, asig

endin
</CsInstruments>
<CsScore>
; sine
f 1 0 32768 10 1

i 1 0 5 220 ;1
i 1 1 4 440 ;2
i 1 2 3 880 ;3, limit is reached
i 1 3 2 1320 ;is not played
i 1 4 1 1760 ;is not played
e
</CsScore>
</CsoundSynthesizer>
```

Its output should contain messages like these:

```
WARNING: cannot allocate last note because it exceeds instr maxalloc
```

## See Also

*cpuprc*, *prealloc*

## Credits

Author: Gabriel Maldonado  
Italy  
July 1999

New in Csound version 3.57

# max\_k

max\_k — Local maximum (or minimum) value of an incoming asig signal

## Description

*max\_k* outputs the local maximum (or minimum) value of the incoming *asig* signal, checked in the time interval between *ktrig* has become true twice.

## Syntax

```
knumkout max_k asig, ktrig, itype
```

## Initialization

*itype* - *itype* determinates the behaviour of *max\_k* (see below)

## Performance

*asig* - incoming (input) signal

*ktrig* - trigger signal

*max\_k* outputs the local maximum (or minimum) value of the incoming *asig* signal, checked in the time interval between *ktrig* has become true twice. *itype* determinates the behaviour of *max\_k*:

- 1 - absolute maximum (sign of negative values is changed to positive before evaluation).
- 2 - actual maximum.
- 3 - actual minimum.
- 4 - calculate average value of *asig* in the time interval since the last trigger.

This opcode can be useful in several situations, for example to implement a vu-meter.

## Examples

Here is an example of the *max\_k* opcode. It uses the file *max\_k.csd* [examples/max\_k.csd].

### Example 525. Example of the max\_k opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -Ma    ;;realtime audio out and midi in (on all inputs)
;-iadc      ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o max_k.wav -W ;; for file output any platform
```

```
</CsOptions>
<CsInstruments>

  sr = 44100
  ksmps = 32
  nchnls = 2
  0dbfs = 1

  FLpanel "This Panel contains VU-meter",300,100
  gk1,gih1 FLslider "VU-meter", 0,1,0,1, -1, 250,30, 30,30
  FLsetColor2 50, 50, 255, gih1
  FLpanel_end
  FLrun

  gal init 0

  instr 1

  kenv linsegr 0,.5,.7,.5,.5,.2,0
  ifreq cpsmidi
  a1 poscil 0dbfs*kenv, ifreq, 1
  gal = gal+a1

  endin

  instr 2

  outs gal, gal
  ktrig metro 25 ;refresh 25 times per second
  kval max_k gal, ktrig, 1
  FLsetVal ktrig, kval, gih1
  gal = 0

  endin

</CsInstruments>
<CsScore>
f1 0 1024 10 1

i2 0 3600
f0 3600

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

Fixed to agree with documentation in 5.15

# maxarray

maxarray — returns the maximum value in an array.

## Description

The *maxarray* opcode returns the maximum value in a k-rate array, and optional its index.

## Syntax

```
kmax [,kindx] maxarray karray
```

## Performance

*kmax* -- variable for result.

*kindx* -- position (index) of result in array.

*karray* -- array for reading.

## Examples

Here is an example of the maxarray opcode. It uses the file *maxarray.csd* [examples/maxarray.csd].

### Example 526. Example of the maxarray opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n
</CsOptions>
<CsInstruments>
ksmps = 32
;example by joachim heintz

          seed          0

instr 1
;create an array with 10 elements
kArr[] init          10
;fill in random numbers and print them out
kIndx = 0
  until kIndx == 10 do
kNum random          -100, 100
kArr[kIndx] = kNum
printf          "kArr[%d] = %10f\n", kIndx+1, kIndx, kNum
kIndx +=          1
  od
;investigate maximum number and print it out
kMax, kMaxIndx maxarray kArr
printf          "Maximum of kArr = %f at index %d\n", kIndx+1, kMax, kMaxIndx
turnoff
endin
```

```
</CsInstruments>  
<CsScore>  
i1 0 0.1  
e  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*minarray*, *sumarray*, *scalearray*,

## Credits

Author: John ffitc  
October 2011

New in Csound version 5.14

Revised in Csound version 6.00 to work on multidimensional arrays

# mclock

mclock — Sends a MIDI CLOCK message.

## Description

Sends a MIDI CLOCK message.

## Syntax

```
mclock ifreq
```

## Initialization

*ifreq* -- clock message frequency rate in Hz

## Performance

Sends a MIDI CLOCK message (0xF8) every  $1/\textit{ifreq}$  seconds. So *ifreq* is the frequency rate of CLOCK message in Hz.

## See Also

*mrtmsg*

## Examples

Here is an example of the mclock opcode. It uses the file *mclock.csd* [examples/mclock.csd].

### Example 527. Example of the mclock opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-Q0    ;;midi out
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;let csound synchronize a sequencer
mclock 24

endin

</CsInstruments>
```

```
<CsScore>  
  
i 1 0 30  
e  
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

# mdelay

mdelay — A MIDI delay opcode.

## Description

A MIDI delay opcode.

## Syntax

```
mdelay kstatus, kchan, kd1, kd2, kdelay
```

## Performance

*kstatus* -- status byte of MIDI message to be delayed

*kchan* -- MIDI channel (1-16)

*kd1* -- first MIDI data byte

*kd2* -- second MIDI data byte

*kdelay* -- delay time in seconds

Each time that *kstatus* is other than zero, *mdelay* outputs a MIDI message to the MIDI out port after *kdelay* seconds. This opcode is useful in implementing MIDI delays. Several instances of *mdelay* can be present in the same instrument with different argument values, so complex and colorful MIDI echoes can be implemented. Further, the delay time can be changed at k-rate.

## Examples

Here is an example of the *mdelay* opcode. It uses the file *mdelay.csd* [examples/mdelay.csd].

### Example 528. Example of the *mdelay* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d          -M0  -Q0
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

; Example by Giorgio Zucco 2007

instr 1 ;Triggered by MIDI notes on channel 1
```

```
kstatus init 0
ifund notnum
ivel veloc

noteondur 1, ifund, ivel, 1

kstatus = kstatus + 1

idel1 = .2
idel2 = .4
idel3 = .6
idel4 = .8

;make four delay lines

mdelay kstatus,1,ifund+2, ivel,idel1
mdelay kstatus,1,ifund+4, ivel,idel2
mdelay kstatus,1,ifund+6, ivel,idel3
mdelay kstatus,1,ifund+8, ivel,idel4

endin

</CsInstruments>
<CsScore>
; Dummy ftable
f 0 60
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Gabriel Maldonado  
Italy  
November 1998

New in Csound version 3.492

# median

median — A median filter, a variant FIR lowpass filter.

## Description

Implementation of a median filter.

## Syntax

```
ares median asig, ksize, imaxsize [, iskip]
```

## Initialization

*imaxsize* -- the maximum size of the window used to select the data.

*iskip* -- initial disposition of internal data space. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*asig* -- input signal to be filtered

*ksize* -- size of the window over which the input is to be filtered. It must not exceed the maximum window size; if it does it is truncated.

*median* is a simple filter that returns the median value of the last *ksize* values. It has a lowpass action. The efficiency decreases as the window size increases.

## Examples

Here is an example of the median opcode. It uses the file *median.csd* [examples/median.csd].

### Example 529. Example of the median opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>

<CsInstruments>
sr = 44100
kr = 441
ksmps = 100
nchnls = 1

instr 1
  a1 oscil 30000, 10, 1
  a2 median a1, 5, 8
  out a2
endin
</CsInstruments>
```

```
<CsScore>  
f1 0 4096 10 1  
i 1 0 0.1  
e  
  
</CsScore>  
  
</CsoundSynthesizer>
```

## Credits

Author: John fitch  
University of Bath  
May 2010

New in Csound version 5.13

# mediank

mediank — A median filter, a variant FIR lowpass filter.

## Description

Implementation of a median filter.

## Syntax

```
kres mediank kin, ksize, imaxsize [, iskip]
```

## Initialization

*imaxsize* -- the maximum size of the window used to select the data.

*iskip* -- initial disposition of internal data space. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*kin* -- krate value to be filtered

*ksize* -- size of the window over which the input is to be filtered. It must not exceed the maximum window size; if it does it is truncated.

*mediank* is a simple filter that returns the median value of the last *ksize* values. It has a lowpass action. The efficiency decreases as the window size increases.

## Examples

Here is an example of the mediank opcode. It uses the file *mediank.csd* [examples/mediank.csd].

### Example 530. Example of the mediank opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-n
</CsOptions>
<CsInstruments>
sr = 44100
kr = 147
ksmps = 300
nchnls = 1

instr 1
  k1 oscil 100, 10, 1
  k2 mediank k1, 5, 8
  printk 0, k2
endin
```

```
</CsInstruments>

<CsScore>
f1 0 4096 10 1
i 1 0 1
e

</CsScore>

</CsoundSynthesizer>
```

## Credits

Author: John ffitch  
University of Bath  
May 2010

New in Csound version 5.13

# metro

metro — Trigger Metronome

## Description

Generate a metronomic signal to be used in any circumstance an isochronous trigger is needed.

## Syntax

```
ktrig metro kfreq [, initphase]
```

## Initialization

*initphase* - initial phase value (in the 0 to 1 range)

## Performance

*ktrig* - output trigger signal

*kfreq* - frequency of trigger bangs in cps

*metro* is a simple opcode that outputs a sequence of isochronous bangs (that is 1 values) each 1/kfreq seconds. Trigger signals can be used in any circumstance, mainly to temporize realtime algorithmic compositional structures.



### Note

*metro* will produce a trigger signal of 1 when its phase is exactly 0 or 1. If you want to skip the initial trigger, use a very small value like 0.00000001.

## Examples

Here is an example of the metro opcode. It uses the file *metro.csd* [examples/metro.csd]

### Example 531. Example of the metro opcode.

```
<CsoundSynthesizer>
<CsOptions>
-odac -B441 -b441
</CsOptions>
<CsInstruments>

sr      =      44100
kr      =      100
ksmps  =      441
nchnls =      2

        instr 1
ktrig metro 0.2
printk2 ktrig
endin
```

```
</CsInstruments>
<CsScore>
i 1 0 20

</CsScore>
</CsoundSynthesizer>
```

Here is another example of the metro opcode. It uses the file *metro-2.csd* [examples/metro-2.csd]

### Example 532. Another example of the metro opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o metro-2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kpch    random    1,20 ;produce values at k-rate
ktrig   metro     10 ;trigger 10 times per second
kval    samphold  kpch, ktrig ;change value whenever ktrig = 1
asig    buzz     1, 220, kval, 1;harmonics
        outs      asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 4096 10 1 ; sine

i 1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

More information on this opcode in the Floss Manuals: [http://en.flossmanuals.net/csound/ch018\\_c-control-structures/](http://en.flossmanuals.net/csound/ch018_c-control-structures/)

## Credits

Written by Gabriel Maldonado.

First Example written by Andrés Cabrera.

New in Csound 5 (Previously available only on CsoundAV)

# mfb

mfb — Mel scale filterbank for spectral magnitudes.

## Description

Applies a Mel scale filterbank to an array containing a series of magnitude bins.

## Syntax

```
kout[] mfb kin[],klow,khigh,ibands
```

## Initialization

*ibands* -- number of bands in the Mel filter bank. Determines the size of the output array.

## Performance

*kout[]* -- output array containing the MFB with *ibands* bands. It will be created if it does not exist.

*kin[]* -- input array containing the input magnitude bins.

*klow* -- lowest frequency (rounded to the nearest bin centre frequency).

*khigh* -- highest lowest frequency (rounded to the nearest bin centre frequency).

## Examples

Here is an example of the mfb opcode. It uses the file *mfb.csd* [examples/mfb.csd].

### Example 533. Example of the mfb opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
ksmps = 64
instr 1

  ifftsize init 1024
  ibins init ifftsize/2
  kIn[] init ifftsize
  kcnt init 0
  p3 = filelen("fox.wav")
  asig diskin "fox.wav"
  kIn shiftin asig
  kcnt += ksmps
  if kcnt == ifftsize then
    kFFT[] = rfft(kIn)
    kPows[] = pows(kFFT)
    kMFB[] = log(mfb(kPows, 300, 8000, 32))
```

```
kmfcc[] = dct(kMFB)
kcnt = 0
kfb = 0
while kfb < 32 do
  printf("mfcc[%d] = %.3f \n", kfb+1, kfb, kmfcc[kfb])
  kfb += 1
od
endif

endin
</CsInstruments>
<CsScore>
i1 0 1
</CsScore>
</CsoundSynthesizer>
```

## See Also

Vectorial opcodes, array opcodes, *dct*

## Credits

Author: Victor Lazzarini  
NUI Maynooth  
2016

New in version 6.08

# midglobal

`midglobal` — An opcode which can be used to implement a remote midi orchestra. This opcode will broadcast the midi events to all the machines involved in the remote concert.

## Description

With the *midremot* and *midglobal* opcodes you are able to perform instruments on remote machines and control them from a master machine. The remote opcodes are implemented using the master/client model. All the machines involved contain the same orchestra but only the master machine contains the information of the midi score. During the performance the master machine sends the midi events to the clients. The *midglobal* opcode sends the events to all the machines involved in the remote concert. These machines are determined by the *midremot* definitions made above the *midglobal* command. To send events to only one machine use *midremot*.

## Syntax

```
midglobal isource, instrnum [,instrnum...]
```

## Initialization

*isource* -- a string that is the intended host computer (e.g. 192.168.0.100). This is the source host which generates the events of the given instrument(s) and sends it to all the machines involved in the remote concert.

*instrnum* -- a list of instrument numbers which will be played on the destination machines

## Examples

See the entry for *midremot* for an example of usage.

## See also

*insglobal*, *insremot*, *midremot*, *remoteport*

## Credits

Author: Simon Schampijer  
2006

New in version 5.03

# midiarp

midiarp — Generates arpeggios based on currently held MIDI notes.

## Description

midiarp constructs arpeggios based currently held MIDI notes. The opcode will output notes in the form of MIDI note numbers, and a metronomic signal that can be used to sequence the notes. Users can choose the rate at which the notes are generated, and may also choose from a set of arpeggio patterns.

## Syntax

```
kMidiNoteNum, kCountermidiarp kRate[, kMode]
```

## Initialization

*kRate* -- sets the rate at which new notes will be generated and controls the frequency of the metronomic pulse signal.

*kMode* -- Optional. Sets the mode of the arpeggio. 0 for up and down, 1, for up, 2 for down, and 3 for random. If left out, it will default to mode 0, up and down.

## Performance

*kMidiNoteNum* -- the current note number in the arpeggio pattern.

*kCounter* -- a metronomic pulse that can be used to trigger playback of the notes in the arpeggio. This signal will output a 1 followed by 0s on each cycle. The frequency is set using the *kRate* input parameter.



### Note

It is important that the instrument which holds the midiarp opcode is not continuously triggered on each new MIDI note. To prevent this from happening use the *massign* opcode, as shown in the example below.

## Example

This example shows how midiarp can be used to trigger arpeggios using a secondary instrument for playback. Instrument 100 listens to incoming MIDI notes, and then triggers playback using instrument 200. *massign* is used to prevent instrument 100 from being triggered each time a new MIDI note is pressed. This examples uses the file *midiarp.csd* [examples/midiarp.csd].

### Example 534. Example of the midiarp opcode

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac -d      -m0d      -M0  -+rtmidi=virtual ;;RT audio I/O with MIDI in
; For Non-realtime ouput leave only the line below:
```

```
; -o midiin.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

massign 1, -1; prevent triggering of instrument with MIDI

instr 100
kMode = 3
kTempo = 6
kNote, kCounter midiarp kTempo

kFilterFreq oscil 2000, .05
;if kCounter is 1 trigger instrument 2 to play
if kCounter==1 then
  event "i", 200, 0, 2, kNote, kFilterFreq+2200
endif

endin

instr 200
kEnv expon .4, p3, .001
aOut vco2 kEnv, cpsmidinn(p4)*2 ;convert note number to cps
aFilter moogladder aOut, p5, 0
outs aFilter, aFilter
endin

</CsInstruments>
<CsScore>
i100 0 1000
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Rory Walsh  
2017

# midic14

*midic14* — Allows a floating-point 14-bit MIDI signal scaled with a minimum and a maximum range.

## Description

Allows a floating-point 14-bit MIDI signal scaled with a minimum and a maximum range.

## Syntax

```
idest midic14 ictlno1, ictlno2, imin, imax [, ifn]
```

```
kdest midic14 ictlno1, ictlno2, kmin, kmax [, ifn]
```

## Initialization

*idest* -- output signal

*ictlno1* -- most-significant byte controller number (0-127)

*ictlno2* -- least-significant byte controller number (0-127)

*imin* -- user-defined minimum floating-point value of output

*imax* -- user-defined maximum floating-point value of output

*ifn* (optional) -- table to be read when indexing is required. Table must be normalized. Output is scaled according to *imin* and *imax* values.

## Performance

*kdest* -- output signal

*kmin* -- user-defined minimum floating-point value of output

*kmax* -- user-defined maximum floating-point value of output

*midic14* (i- and k-rate 14 bit MIDI control) allows a floating-point 14-bit MIDI signal scaled with a minimum and a maximum range. The minimum and maximum values can be varied at k-rate. It can use optional interpolated table indexing. It requires two MIDI controllers as input.



### Note

Please note that the *midic* family of opcodes are designed for MIDI triggered events, and don't require a channel number since they will respond to the same channel as the one that triggered the instrument (see *massign*). However they will crash if called from a score driven event.

## See Also

*ctrl7, ctrl14, ctrl21, inite7, inite14, inite21, midic7, midic21*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct controller number range.

# midic21

`midic21` — Allows a floating-point 21-bit MIDI signal scaled with a minimum and a maximum range.

## Description

Allows a floating-point 21-bit MIDI signal scaled with a minimum and a maximum range.

## Syntax

```
idest midic21 ictlno1, ictlno2, ictlno3, imin, imax [, ifn]
kdest midic21 ictlno1, ictlno2, ictlno3, kmin, kmax [, ifn]
```

## Initialization

*idest* -- output signal

*ictlno1* -- most-significant byte controller number (0-127)

*ictlno2* -- mid-significant byte controller number (0-127)

*ictlno3* -- least-significant byte controller number (0-127)

*imin* -- user-defined minimum floating-point value of output

*imax* -- user-defined maximum floating-point value of output

*ifn* (optional) -- table to be read when indexing is required. Table must be normalized. Output is scaled according to the *imin* and *imax* values.

## Performance

*kdest* -- output signal

*kmin* -- user-defined minimum floating-point value of output

*kmax* -- user-defined maximum floating-point value of output

*midic21* (i- and k-rate 21 bit MIDI control) allows a floating-point 21-bit MIDI signal scaled with a minimum and a maximum range. Minimum and maximum values can be varied at k-rate. It can use optional interpolated table indexing. It requires three MIDI controllers as input.



### Note

Please note that the *midic* family of opcodes are designed for MIDI triggered events, and don't require a channel number since they will respond to the same channel as the one that triggered the instrument (see *massign*). However they will crash if called from a score driven event.

## See Also

*ctrl7, ctrl14, ctrl21, inite7, inite14, inite21, midic7, midic14*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct controller number range.

# midic7

*midic7* — Allows a floating-point 7-bit MIDI signal scaled with a minimum and a maximum range.

## Description

Allows a floating-point 7-bit MIDI signal scaled with a minimum and a maximum range.

## Syntax

```
idest midic7 ictlno, imin, imax [, ifn]
```

```
kdest midic7 ictlno, kmin, kmax [, ifn]
```

## Initialization

*idest* -- output signal

*ictlno* -- MIDI controller number (0-127)

*imin* -- user-defined minimum floating-point value of output

*imax* -- user-defined maximum floating-point value of output

*ifn* (optional) -- table to be read when indexing is required. Table must be normalized. Output is scaled according to the *imin* and *imax* values.

## Performance

*kdest* -- output signal

*kmin* -- user-defined minimum floating-point value of output

*kmax* -- user-defined maximum floating-point value of output

*midic7* (i- and k-rate 7 bit MIDI control) allows a floating-point 7-bit MIDI signal scaled with a minimum and a maximum range. It also allows optional non-interpolated table indexing. In *midic7* minimum and maximum values can be varied at k-rate.



### Note

Please note that the *midic* family of opcodes are designed for MIDI triggered events, and don't require a channel number since they will respond to the same channel as the one that triggered the instrument (see *massign*). However they will crash if called from a score driven event.

## Examples

Here is an example of the *midic7* opcode. It uses the file *midic7.csd* [examples/midic7.csd]

**Example 535. Example of the *midic7* opcode.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0      ;;realtime audio out and realtime midi in
;-iadc      ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o midic7.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
; This example expects MIDI controller input on channel 1
; run, play a note and move your midi controller 7 to see result

imax = 1
imin = 0
ichan = 1
ictlno= 7 ; = midi volume

kamp midic7 ictlno, imin, imax
printk2 kamp
asig oscili kamp, 220, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
; no score events allowed
f 0 20 ;20 sec. for real-time MIDI events
f 1 0 4096 10 1 ;sine wave

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*ctrl7, ctrl14, ctrl21, inite7, inite14, inite21, midic14, midic21*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct controller number range.

# midichannelaftertouch

midichannelaftertouch — Gets a MIDI channel's aftertouch value.

## Description

*midichannelaftertouch* is designed to simplify writing instruments that can be used interchangeably for either score or MIDI input, and to make it easier to adapt instruments originally written for score input to work with MIDI input.

In general, it should be possible to write instrument definitions that work identically with both scores and MIDI, including both MIDI files and real-time MIDI input, without using any conditional statements, and that take full advantage of MIDI voice messages.

Note that correlating Csound instruments with MIDI channel numbers is done using the *massign* opcode for real-time performance,. For file-driven performance, instrument numbers default to MIDI channel number + 1, but the defaults are overridden by any MIDI program change messages in the file.

## Syntax

```
midichannelaftertouch xchannelaftertouch [, ilow] [, ihigh]
```

## Initialization

*ilow* (optional) -- optional low value after rescaling, defaults to 0.

*ihigh* (optional) -- optional high value after rescaling, defaults to 127.

## Performance

*xchannelaftertouch* -- returns the MIDI channel aftertouch during MIDI activation, remains unchanged otherwise.

If the instrument was activated by MIDI input, the opcode overwrites the value of *xchannelaftertouch* with the corresponding value from MIDI input. If the instrument was *NOT* activated by MIDI input, the value of *xchannelaftertouch* remains unchanged.

This enables score p-fields to receive MIDI input data during MIDI activation, and score values otherwise.



### **Adapting a score-activated Csound instrument.**

See the *MIDI interop opcodes* section for details on adapting score driven instruments for MIDI or vice-versa.

## Examples

Here is an example of the *midichannelaftertouch* opcode. It uses the file *midichannelaftertouch.csd* [examples/midichannelaftertouch.csd].

### **Example 536. Example of the midichannelaftertouch opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -M1 ;;realtime audio out and midi in
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o midichannelaftertouch.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

midinoteoncps p4, p5 ;puts MIDI key translated to cycles per second into p4, and MIDI velocity into
kafter init 127 ;full open
midichannelaftertouch kafter
printk2 kafter ;display the key value when it changes and when key is pressed

kvel = p5/127 ;scale midi velocity to 0-1
kenv madsr 0.5, 0.8, 0.8, 0.5 ;amplitude envelope multiplied by
ain pluck kenv*kvel, p4, p4, 2, 1 ;velocity value
asig moogvcf2 ain, 200+(kafter*40), .5 ;scale value of aftertouch and control filter frequency
outs asig, asig ;base freq of filter = 200 Hz

endin
</CsInstruments>
<CsScore>
f 0 30 ;runs 30 seconds
f 2 0 4096 10 1

i 1 0 2 440 100 ; play these notes from score as well
i 1 + 2 1440 100
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like:

```
i1 127.00000
i1 20.00000
i1 44.00000
```

## See Also

*midicontrolchange, mididefault, midinoteoff, midinoteoncps, midinoteonkey, midinoteonoct, midinoteon-  
pch, midipitchbend, midipolyaftertouch, midiprogramchange*

## Credits

Author: Michael Gogins

New in version 4.20

# midichn

midichn — Returns the MIDI channel number from which the note was activated.

## Description

*midichn* returns the MIDI channel number (1 - 16) from which the note was activated. In the case of score notes, it returns 0.

## Syntax

```
ichn midichn
```

## Initialization

*ichn* -- channel number. If the current note was activated from score, it is set to zero.

## Examples

Here is a simple example of the midichn opcode. It uses the file *midichn.csd* [examples/midichn.csd].

### Example 537. Example of the midichn opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -M0 -+rtmidi=virtual ;; midi file input
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
;change channel on virtual midi keyboard
i1 midichn
  print i1

endin
</CsInstruments>
<CsScore>
f 0 20 ;runs for 20 seconds

e
</CsScore>
</CsoundSynthesizer>
```

Here is an advanced example of the midichn opcode. It uses the file *midichn\_advanced.csd* [examples/midichn\_advanced.csd].

Don't forget that you must include the *-F flag* when using an external MIDI file like “midichn\_advanced.mid”.

### Example 538. An advanced example of the midichn opcode.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -F midichn_advanced.mid ;;realtime audio out with MIDI file input
; For Non-realtime ouput leave only the line below:
; -o midichn_advanced.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

massign 1, 1 ; all channels use instr 1
massign 2, 1
massign 3, 1
massign 4, 1
massign 5, 1
massign 6, 1
massign 7, 1
massign 8, 1
massign 9, 1
massign 10, 1
massign 11, 1
massign 12, 1
massign 13, 1
massign 14, 1
massign 15, 1
massign 16, 1

gicnt = 0 ; note counter

instr 1

gicnt = gicnt + 1 ; update note counter
kcnt init gicnt ; copy to local variable
ichn midichn ; get channel number
istime times ; note-on time

if (ichn > 0.5) goto l2 ; MIDI note
printks "note %.0f (time = %.2f) was activated from the score\\n", \
3600, kcnt, istime
goto l1
l2:
printks "note %.0f (time = %.2f) was activated from channel %.0f\\n", \
3600, kcnt, istime, ichn
l1:

icps cpsmidi ; convert midi note to pitch
kenv madsr 0.1, 0, 0.8, 0.9
asig pluck kenv, icps, icps, 1, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
t 0 60 ;beats per minute

f 0 8 ;stay active for 8 seconds

```

```
f 1 0 4096 10 1 ;sine
```

```
e  
</CsScore>  
</CsoundSynthesizer>
```

Its output should include lines like:

```
note 1 (time = 0.00) was activated from channel 1  
note 2 (time = 2.00) was activated from channel 4  
note 3 (time = 3.00) was activated from channel 2  
note 4 (time = 5.00) was activated from channel 3
```

## See Also

*pgmassign*

## Credits

Author: Istvan Varga  
May 2002

New in version 4.20

# midicontrolchange

midicontrolchange — Gets a MIDI control change value.

## Description

*midicontrolchange* is designed to simplify writing instruments that can be used interchangeably for either score or MIDI input, and to make it easier to adapt instruments originally written for score input to work with MIDI input.

In general, it should be possible to write instrument definitions that work identically with both scores and MIDI, including both MIDI files and real-time MIDI input, without using any conditional statements, and that take full advantage of MIDI voice messages.

Note that correlating Csound instruments with MIDI channel numbers is done using the *massign* opcode for real-time performance. For file-driven performance, instrument numbers default to MIDI channel number + 1, but the defaults are overridden by any MIDI program change messages in the file.

## Syntax

```
midicontrolchange xcontroller, xcontrollervalue [, ilow] [, ihigh]
```

## Initialization

*ilow* (optional) -- optional low value after rescaling, defaults to 0.

*ihigh* (optional) -- optional high value after rescaling, defaults to 127.

## Performance

*xcontroller* -- specifies a MIDI controller number (0-127).

*xcontrollervalue* -- returns the value of the MIDI controller during MIDI activation, remains unchanged otherwise.

If the instrument was activated by MIDI input, the opcode overwrites the values of the *xcontroller* and *xcontrollervalue* with the corresponding values from MIDI input. If the instrument was *NOT* activated by MIDI input, the values of *xcontroller* and *xcontrollervalue* remain unchanged.

This enables score p-fields to receive MIDI input data during MIDI activation, and score values otherwise.



### Adapting a score-activated Csound instrument.

See the *MIDI interop opcodes* section for details on adapting score driven instruments for MIDI or vice-versa.

## Examples

Here is an example of the midicontrolchange opcode. It uses the file *midicontrolchange.csd* [examples/midicontrolchange.csd]

**Example 539. Example of the midicontrolchange opcode.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0 ;;realtime audio out and midi in
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o midicontrolchange.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
; use slider of contr. 7 of virtual keyboard
kcont init 1 ; max. volume
midicontrolchange 7, kcont, 0, 1; use controller 7, scaled between 0 and 1
printk2 kcont ; Display the key value when it changes and key is pressed

kenv madsr 0.5, 0.8, 0.8, 0.5 ; envelope multiplied by
asig pluck kenv*kcont, 220, 220, 2, 1 ; value of controller 7
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 0 30
f 2 0 4096 10 1

i 1 10 2 ; play a note from score as well
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*midichannelaftertouch, mididefault, midinoteoff, midinoteoncps, midinoteonkey, midinoteonoct, midino-  
teonpch, midipitchbend, midipolyaftertouch, midiprogramchange*

## Credits

Author: Michael Gogins

New in version 4.20

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# midictrl

midictrl — Get the current value (0-127) of a specified MIDI controller.

## Description

Get the current value (0-127) of a specified MIDI controller.

## Syntax

```
ival midictrl inum [, imin] [, imax]
kval midictrl inum [, imin] [, imax]
```

## Initialization

*inum* -- MIDI controller number (0-127)

*imin*, *imax* -- set minimum and maximum limits on values obtained.

## Performance

Get the current value (0-127) of a specified MIDI controller.

## Warning

*midictrl* should only be used in notes that were triggered from MIDI, so that an associated channel number is available. For notes activated from the score, line events, or orchestra, the *ctrl7* opcode that takes an explicit channel number should be used instead.

## Examples

Here is an example of the midictrl opcode. It uses the file *midictrl.csd* [examples/midictrl.csd]

### Example 540. Example of the midictrl opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0   ;;realtime audio out and realtime midi in
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o midictrl.wav -W   ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
```

```
icps cpsmidi
iamp ampmidi .5
ips midictrl 9, 10, 500 ;controller 9

kenv madsr 0.5, 0, 1, 0.5
asig pluck kenv, icps, ips, 2, 1 ;change tone color
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 2 0 4096 10 1 ;sine wave
; no score events allowed
f0 30 ;runs 30 seconds
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*aftouch, ampmidi, cpsmidi, cpsmidib, notnum, octmidi, octmidib, pchbend, pchmidi, pchmidib, veloc*

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

Thanks goes to Rasmus Ekman for pointing out the correct controller number range.

# mididefault

mididefault — Changes values, depending on MIDI activation.

## Description

*mididefault* is designed to simplify writing instruments that can be used interchangeably for either score or MIDI input, and to make it easier to adapt instruments originally written for score input to work with MIDI input.

In general, it should be possible to write instrument definitions that work identically with both scores and MIDI, including both MIDI files and real-time MIDI input, without using any conditional statements, and that take full advantage of MIDI voice messages.

Note that correlating Csound instruments with MIDI channel numbers is done using the *massign* opcode for real-time performance,. For file-driven performance, instrument numbers default to MIDI channel number + 1, but the defaults are overridden by any MIDI program change messages in the file.

## Syntax

```
mididefault xdefault, xvalue
```

## Performance

*xdefault* -- specifies a default value that will be used during MIDI activation.

*xvalue* -- overwritten by *xdefault* during MIDI activation, remains unchanged otherwise.

If the instrument was activated by MIDI input, the opcode will overwrite the value of *xvalue* with the value of *xdefault*. If the instrument was *NOT* activated by MIDI input, *xvalue* will remain unchanged.

This enables score pfields to receive a default value during MIDI activation, and score values otherwise.



### Adapting a score-activated Csound instrument.

See the *MIDI interop opcodes* section for details on adapting score driven instruments for MIDI or vice-versa.

## Examples

Here is an example of the mididefault opcode. It uses the file *mididefault.csd* [examples/mididefault.csd].

### Example 541. Example of the mididefault opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -M1 ;;realtime audio out and midi in
;-iadc ;;uncomment -iadc if realtime audio input is needed too
```

```
; For Non-realtime ouput leave only the line below:
; -o mididefault.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

mididefault 60, p3 ;ensures that a MIDI-activated instrument will have a positive p3 field
midinoteoncps p4, p5 ;puts MIDI key translated to cycles per second into p4, and MIDI velocity into p5

kvel = p5/127 ;scale midi velocity to 0-1
kenv madsr 0.5, 0.8, 0.8, 0.5 ;amplitude envelope multiplied by
asig pluck kenv*kvel, p4, p4, 2, 1 ;velocity
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 0 30 ;runs for 30 seconds
f 2 0 4096 10 1

i 1 0 2 440 100 ; play these notes from score as well
i 1 + 2 1440 100
e

</CsScore>
</CsSoundSynthesizer>
```

## See Also

*midichannelaftertouch, midicontrolchange, midinoteoff, midinoteoncps, midinoteonkey, midinoteonoct, midinoteonpch, midipitchbend, midipolyaftertouch, midiprogramchange*

## Credits

Author: Michael Gogins

New in version 4.20

## midiin

`midiin` — Returns a generic MIDI message received by the MIDI IN port.

### Description

Returns a generic MIDI message received by the MIDI IN port

### Syntax

```
kstatus, kchan, kdata1, kdata2 midiin
```

### Performance

*kstatus* -- the type of MIDI message. Can be:

- 128 (note off)
- 144 (note on)
- 160 (polyphonic aftertouch)
- 176 (control change)
- 192 (program change)
- 208 (channel aftertouch)
- 224 (pitch bend)
- 0 if no MIDI message are pending in the MIDI IN buffer

*kchan* -- MIDI channel (1-16)

*kdata1, kdata2* -- message-dependent data values

*midiin* has no input arguments, because it reads at the MIDI in port implicitly. It works at k-rate. Normally (i.e., when no messages are pending) *kstatus* is zero, only when MIDI data are present in the MIDI IN buffer, is *kstatus* set to the type of the relevant messages.



#### Note

Be careful when using *midiin* in low numbered instruments, since a MIDI note will launch additional instances of the instrument, resulting in duplicate events and weird behaviour. Use *massign* to direct MIDI note on messages to a different instrument or to disable triggering of instruments from MIDI.

### Examples

Here is an example of the *midiin* opcode. It uses the file *midiin.csd* [examples/midiin.csd].

**Example 542. Example of the midiin opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>

<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc      -d          -M0  -+rtmidi=virtual  ;;RT audio I/O with MIDI in
; For Non-realtime ouput leave only the line below:
; -o midiin.wav -W ;; for file output any platform
</CsOptions>

<CsInstruments>

sr          = 44100
ksmps      = 10
nchnls     = 1

; Example by schwaahed 2006

      massign          0, 130 ; make sure that all channels
      pgmassign       0, 130 ; and programs are assigned to test instr

instr 130

knotelength  init  0
knoteontime  init  0

kstatus, kchan, kdata1, kdata2          midiin

if (kstatus == 128) then
knoteofftime  times
knotelength   =   knoteofftime - knoteontime
printks "kstatus= %d, kchan = %d, \\tnote# = %d, velocity = %d \\tNote OFF\\t%f\\n", 0, kstatus, kchan, kdata1, kdata2

elseif (kstatus == 144) then
knoteontime   times
printks "kstatus= %d, kchan = %d, \\tnote# = %d, velocity = %d \\tNote ON\\t%f\\n", 0, kstatus, kchan, kdata1, kdata2

elseif (kstatus == 160) then
printks "kstatus= %d, kchan = %d, \\tkdata1 = %d, kdata2 = %d \\tPolyphonic Aftertouch\\n", 0, kstatus, kchan, kdata1, kdata2

elseif (kstatus == 176) then
printks "kstatus= %d, kchan = %d, \\t CC = %d, value = %d \\tControl Change\\n", 0, kstatus, kchan, kdata1, kdata2

elseif (kstatus == 192) then
printks "kstatus= %d, kchan = %d, \\tkdata1 = %d, kdata2 = %d \\tProgram Change\\n", 0, kstatus, kchan, kdata1, kdata2

elseif (kstatus == 208) then
printks "kstatus= %d, kchan = %d, \\tkdata1 = %d, kdata2 = %d \\tChannel Aftertouch\\n", 0, kstatus, kchan, kdata1, kdata2

elseif (kstatus == 224) then
printks "kstatus= %d, kchan = %d, \\t ( data1 , kdata2 ) = ( %d, %d )\\tPitch Bend\\n", 0, kstatus, kchan, kdata1, kdata2

endif

      endin

</CsInstruments>

<CsScore>

```

```
i130 0 3600  
e  
</CsScore>  
  
</CsoundSynthesizer>
```

## Credits

Author: Gabriel Maldonado  
Italy  
1998

New in Csound version 3.492

## midfilestatus

midfilestatus — Returns the playback status of MIDI file input.

### Description

Returns the current playback status at k-rate, of the input MIDI file, 1 if file is playing, 0 if the end-of-the file has been reached.

### Syntax

```
ksig midfilestatus
```

### Credits

Author: Victor Lazzarini  
March 2006  
New in Csound6

# midinoteoff

midinoteoff — Gets a MIDI noteoff value.

## Description

*midinoteoff* is designed to simplify writing instruments that can be used interchangeably for either score or MIDI input, and to make it easier to adapt instruments originally written for score input to work with MIDI input.

In general, it should be possible to write instrument definitions that work identically with both scores and MIDI, including both MIDI files and real-time MIDI input, without using any conditional statements, and that take full advantage of MIDI voice messages.

Note that correlating Csound instruments with MIDI channel numbers is done using the *massign* opcode for real-time performance. For file-driven performance, instrument numbers default to MIDI channel number + 1, but the defaults are overridden by any MIDI program change messages in the file.

## Syntax

```
midinoteoff xkey, xvelocity
```

## Performance

*xkey* -- returns MIDI key during MIDI activation, remains unchanged otherwise.

*xvelocity* -- returns MIDI velocity during MIDI activation, remains unchanged otherwise.

If the instrument was activated by MIDI input, the opcode overwrites the values of the *xkey* and *xvelocity* with the corresponding values from MIDI input. If the instrument was *NOT* activated by MIDI input, the values of *xkey* and *xvelocity* remain unchanged.

This enables score p-fields to receive MIDI input data during MIDI activation, and score values otherwise.



### Adapting a score-activated Csound instrument.

See the *MIDI interop opcodes* section for details on adapting score driven instruments for MIDI or vice-versa.

## Examples

Here is an example of the midinoteoff opcode. It uses the file *midinoteoff.csd* [examples/midinoteoff.csd].

### Example 543. Example of the midinoteoff opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform
```

```
-odac -M1 -Q1 ;;;realtime audio out and midi in and midi out
;-iadc ;;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o midinoteoff.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;uses external midi device

kkey init 0 ;initialize key number
kvel init 0 ;initialize velocity
midinoteoff kkey,kvel ;MIDI noteoff value
printk2 kvel ;display noteoff value
midion 1, kkey, kvel ;sent note to external device

endin
</CsInstruments>
<CsScore>
f 0 30 ;runs for 30 seconds

i 1 0 2 62 ; play these notes from score as well
i 1 + 2 65
e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like:

```
i1 60.00000
i1 76.00000
```

## See Also

*midichannelaftertouch, midicontrolchange, mididefault, midinoteoncps, midinoteonkey, midinoteonoct, midinoteonpch, midipitchbend, midipolyaftertouch, midiprogramchange*

## Credits

Author: Michael Gogins

New in version 4.20

# midinoteoncps

midinoteoncps — Gets a MIDI note number as a cycles-per-second frequency.

## Description

*midinoteoncps* is designed to simplify writing instruments that can be used interchangeably for either score or MIDI input, and to make it easier to adapt instruments originally written for score input to work with MIDI input.

In general, it should be possible to write instrument definitions that work identically with both scores and MIDI, including both MIDI files and real-time MIDI input, without using any conditional statements, and that take full advantage of MIDI voice messages.

Note that correlating Csound instruments with MIDI channel numbers is done using the *massign* opcode for real-time performance,. For file-driven performance, instrument numbers default to MIDI channel number + 1, but the defaults are overridden by any MIDI program change messages in the file.

## Syntax

```
midinoteoncps xcps, xvelocity
```

## Performance

*xcps* -- returns MIDI key translated to cycles per second during MIDI activation, remains unchanged otherwise.

*xvelocity* -- returns MIDI velocity during MIDI activation, remains unchanged otherwise.

If the instrument was activated by MIDI input, the opcode overwrites the values of *xcps* and *xvelocity* with the corresponding values from MIDI input. If the instrument was *NOT* activated by MIDI input, the values of *xcps* and *xvelocity* remain unchanged.

This enables score p-fields to receive MIDI input data during MIDI activation, and score values otherwise.



### Adapting a score-activated Csound instrument.

See the *MIDI interop opcodes* section for details on adapting score driven instruments for MIDI or vice-versa.

## Examples

Here is an example of the *midinoteoncps* opcode. It uses the file *midinoteoncps.csd* [examples/midinoteoncps.csd].

### Example 544. Example of the midinoteoncps opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
```

```
<CsOptions>
; Select audio/midi flags here according to platform
-odac -M1 ;;realtime audio out and midi in
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o midinoteoncps.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

midinoteoncps p4, p5 ;puts MIDI key translated to cycles per second into p4, and MIDI velocity into

print p4 ;display the key value when it changes and when key is pressed
kvel = p5/127 ;scale midi velocity to 0-1
kenv madsr 0.5, 0.8, 0.8, 0.5 ;amplitude envelope multiplied by
asig pluck kenv*kvel, p4, p4, 2, 1 ;velocity value
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 0 30 ;runs for 30 seconds
f 2 0 4096 10 1

i 1 0 2 440 100 ; play these notes from score as well
i 1 + 2 1440 100
e

</CsScore>
</CsSoundSynthesizer>
```

Its output should include lines like:

```
i1 261.62561
i1 440.00006
```

## See Also

*midichannelaftertouch, midicontrolchange, mididefault, midinoteoff, midinoteonkey, midinoteonoct, midinoteonpch, midipitchbend, midipolyaftertouch, midiprogramchange*

## Credits

Author: Michael Gogins

New in version 4.20

# midinoteonkey

midinoteonkey — Gets a MIDI note number value.

## Description

*midinoteonkey* is designed to simplify writing instruments that can be used interchangeably for either score or MIDI input, and to make it easier to adapt instruments originally written for score input to work with MIDI input.

In general, it should be possible to write instrument definitions that work identically with both scores and MIDI, including both MIDI files and real-time MIDI input, without using any conditional statements, and that take full advantage of MIDI voice messages.

Note that correlating Csound instruments with MIDI channel numbers is done using the *massign* opcode for real-time performance. For file-driven performance, instrument numbers default to MIDI channel number + 1, but the defaults are overridden by any MIDI program change messages in the file.

## Syntax

```
midinoteonkey xkey, xvelocity
```

## Performance

*xkey* -- returns MIDI key during MIDI activation, remains unchanged otherwise.

*xvelocity* -- returns MIDI velocity during MIDI activation, remains unchanged otherwise.

If the instrument was activated by MIDI input, the opcode overwrites the values of *xkey* and *xvelocity* with the corresponding values from MIDI input. If the instrument was *NOT* activated by MIDI input, the values of *xkey* and *xvelocity* remain unchanged.

This enables score p-fields to receive MIDI input data during MIDI activation, and score values otherwise.



### Adapting a score-activated Csound instrument.

See the *MIDI interop opcodes* section for details on adapting score driven instruments for MIDI or vice-versa.

## Examples

Here is an example of the midinoteonkey opcode. It uses the file *midinoteonkey.csd* [examples/midinoteonkey.csd].

### Example 545. Example of the midinoteonkey opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>
```

```
; Select audio/midi flags here according to platform
-odac -M1 -Q1 ;;realtime audio out and midi in and midi out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o midinoteonkey.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;uses external midi device

kkey init 0 ;initialize key number
kvel init 0 ;initialize velocity
midinoteonkey kkey, kvel ;MIDI note number value
printk2 kkey ;display the key value when it changes and when key is pressed
midion 1, kkey, kvel ;sent note to external device

endin
</CsInstruments>
<CsScore>
f 0 30 ;runs for 30 seconds

i 1 0 2 62 ; play these notes from score as well
i 1 + 2 65
e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like:

```
i1 60.00000
i1 69.00000
```

## See Also

*midichannelaftertouch, midicontrolchange, mididefault, midinoteoff, midinoteoncps, midinoteonoct, midinoteonpch, midipitchbend, midipolyaftertouch, midiprogramchange*

## Credits

Author: Michael Gogins

New in version 4.20

# midinoteonct

midinoteonct — Gets a MIDI note number value as octave-point-decimal value.

## Description

*midinoteonct* is designed to simplify writing instruments that can be used interchangeably for either score or MIDI input, and to make it easier to adapt instruments originally written for score input to work with MIDI input.

In general, it should be possible to write instrument definitions that work identically with both scores and MIDI, including both MIDI files and real-time MIDI input, without using any conditional statements, and that take full advantage of MIDI voice messages.

Note that correlating Csound instruments with MIDI channel numbers is done using the *massign* opcode for real-time performance. For file-driven performance, instrument numbers default to MIDI channel number + 1, but the defaults are overridden by any MIDI program change messages in the file.

## Syntax

```
midinoteonct xoct, xvelocity
```

## Performance

*xoct* -- returns MIDI key translated to linear octaves during MIDI activation, remains unchanged otherwise.

*xvelocity* -- returns MIDI velocity during MIDI activation, remains unchanged otherwise.

If the instrument was activated by MIDI input, the opcode overwrites the values of *xoct* and *xvelocity* with the corresponding value from MIDI input. If the instrument was *NOT* activated by MIDI input, the values of *xoct* and *xvelocity* remain unchanged.

This enables score p-fields to receive MIDI input data during MIDI activation, and score values otherwise.



### Adapting a score-activated Csound instrument.

See the *MIDI interop opcodes* section for details on adapting score driven instruments for MIDI or vice-versa.

## Examples

Here is an example of the midinoteonct opcode. It uses the file *midinoteonct.csd* [examples/midinoteonct.csd].

### Example 546. Example of the midinoteonct opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
```

```

; Select audio/midi flags here according to platform
-odac -M1 ;;;realtime audio out and midi in
;-iadc ;;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o midinoteonct.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

midinoteonct p4, p5 ;gets a MIDI note number value as octave-point-decimal value into p4, and MIDI v

print p4 ;display the key value when it changes and when key is pressed
kvel = p5/127 ;scale midi velocity to 0-1
ioct = p4
icps = cpsoct(ioct) ;convert octave-point-decimal value into Hz
kenv madsr 0.5, 0.8, 0.8, 0.5 ;amplitude envelope multiplied by
asig pluck kenv*kvel, icps, icps, 2, 1 ;velocity value
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 0 30 ;runs for 30 seconds
f 2 0 4096 10 1

i 1 0 2 8.000 100 ; play these notes from score as well
i 1 + 2 8.917 100
e

</CsScore>
</CsSoundSynthesizer>

```

Its output should include lines like:

```

i1      8.00000
i1      8.91700
i1      9.33333

```

## See Also

*midichannelaftertouch*, *midicontrolchange*, *mididefault*, *midinoteoff*, *midinoteoncps*, *midinoteonkey*, *midinoteonpch*, *midipitchbend*, *midipolyaftertouch*, *midiprogramchange*

## Credits

Author: Michael Gogins

New in version 4.20

# midinoteonpch

midinoteonpch — Gets a MIDI note number as a pitch-class value.

## Description

*midinoteonpch* is designed to simplify writing instruments that can be used interchangeably for either score or MIDI input, and to make it easier to adapt instruments originally written for score input to work with MIDI input.

In general, it should be possible to write instrument definitions that work identically with both scores and MIDI, including both MIDI files and real-time MIDI input, without using any conditional statements, and that take full advantage of MIDI voice messages.

Note that correlating Csound instruments with MIDI channel numbers is done using the *massign* opcode for real-time performance. For file-driven performance, instrument numbers default to MIDI channel number + 1, but the defaults are overridden by any MIDI program change messages in the file.

## Syntax

```
midinoteonpch xpch, xvelocity
```

## Performance

*xpch* -- returns MIDI key translated to octave.pch during MIDI activation, remains unchanged otherwise.

*xvelocity* -- returns MIDI velocity during MIDI activation, remains unchanged otherwise.

If the instrument was activated by MIDI input, the opcode overwrites the values of *xpch* and *xvelocity* with the corresponding value from MIDI input. If the instrument was *NOT* activated by MIDI input, the values of *xpch* and *xvelocity* remain unchanged.

This enables score p-fields to receive MIDI input data during MIDI activation, and score values otherwise.



### Adapting a score-activated Csound instrument.

See the *MIDI interop opcodes* section for details on adapting score driven instruments for MIDI or vice-versa.

## Examples

Here is an example of the midinoteonpch opcode. It uses the file *midinoteonpch.csd* [examples/midinoteonpch.csd].

### Example 547. Example of the midinoteonpch opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>
```

```
; Select audio/midi flags here according to platform
-odac -M1 ;;;realtime audio out and midi in
;-iadc ;;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o midinoteonpch.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

midinoteonpch p4, p5 ;gets a MIDI note number value as octave-point-decimal value into p4, and MIDI

print p4 ;display the pitch value when it changes and when key is pressed
kvel = kvel/127 ;scale midi velocity to 0-1
ipch = p4
icps = cpspch(ipch) ;convert octave-point-decimal value into Hz
kenv madsr 0.5, 0.8, 0.8, 0.5 ;amplitude envelope multiplied by
asig pluck kenv*kvel, icps, icps, 2, 1 ;velocity value
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 0 30 ;runs for 30 seconds
f 2 0 4096 10 1

i 1 0 2 8.09 100 ; play these notes from score as well
i 1 + 2 9.05 100
e

</CsScore>
</CsSoundSynthesizer>
```

Its output should include lines like:

```
i1      8.09000
i1      9.05000
```

## See Also

*midichannelaftertouch, midicontrolchange, mididefault, midinoteoff, midinoteoncps, midinoteonkey, midinoteonoct, midipitchbend, midipolyaftertouch, midiprogramchange*

## Credits

Author: Michael Gogins

New in version 4.20

# midion2

midion2 — Sends noteon and noteoff messages to the MIDI OUT port.

## Description

Sends noteon and noteoff messages to the MIDI OUT port when triggered by a value different than zero.

## Syntax

```
midion2 kchn, knum, kvel, ktrig
```

## Performance

*kchn* -- MIDI channel (1-16)

*knun* -- MIDI note number (0-127)

*kvel* -- note velocity (0-127)

*ktrig* -- trigger input signal (normally 0)

Similar to *midion*, this opcode sends noteon and noteoff messages to the MIDI out port, but only when *ktrig* is non-zero. This opcode is can work together with the output of the *trigger* opcode.

## Examples

Here is an example of the midion2 opcode. It uses the file *midion2.csd* [examples/midion2.csd].

### Example 548. Example of the midion2 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-M0 -Q1 ;;midi in and midi out
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kcps line 3, p3, .1
klf lfo 1, kcps, 3 ;use a unipolar square to trigger
ktr trigger klf, 1, 1 ;from 3 times to .1 time per sec.
midion2 1, 60, 100, ktr

endin
</CsInstruments>
```

```
<CsScore>  
  
i 1 0 20  
e  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*moscil, midion, noteon, noteoff, noteondur, noteondur2*

## Credits

Author: Gabriel Maldonado  
Italy  
1998

New in Csound version 3.492

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# midion

midion — Generates MIDI note messages at k-rate.

## Description

Generates MIDI note messages at k-rate.

## Syntax

```
midion kchn, knum, kvel
```

## Performance

*kchn* -- MIDI channel number (1-16)

*knum* -- note number (0-127)

*kvel* -- velocity (0-127)

*midion* (k-rate note on) plays MIDI notes with current *kchn*, *knum* and *kvel*. These arguments can be varied at k-rate. Each time the MIDI converted value of any of these arguments changes, last MIDI note played by current instance of *midion* is immediately turned off and a new note with the new argument values is activated. This opcode, as well as *moscil*, can generate very complex melodic textures if controlled by complex k-rate signals.

Any number of *midion* opcodes can appear in the same Csound instrument, allowing a counterpoint-style polyphony within a single instrument.

## Examples

Here is a simple example of the *midion* opcode. It uses the file *midion\_simple.csd* [examples/midion\_simple.csd].

### Example 549. Simple Example of the *midion* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

This example generates a minor chord over every note received on the MIDI input. It generates MIDI notes on csound's MIDI output, so be sure to connect something.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      -M0  -Q1  ;;RT audio I/O with MIDI in
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
```

```

nchnls = 2

; Example by Giorgio Zucco 2007

instr 1 ;Triggered by MIDI notes on channel 1

  ifund notnum
  ivel veloc

  knote1 init ifund
  knote2 init ifund + 3
  knote3 init ifund + 5

  ;minor chord on MIDI out channel 1
  ;Needs something plugged to csound's MIDI output
  midion 1, knote1,ivel
  midion 1, knote2,ivel
  midion 1, knote3,ivel

endin

</CsInstruments>
<CsScore>
; Dummy ftable
f0 60
</CsScore>
</CsoundSynthesizer>

```

Here is another example of the midion opcode. It uses the file *midion\_scale.csd* [examples/midion\_scale.csd].

### Example 550. Example of the midion opcode to generate random notes from a scale.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

This example generates random notes from a given scale for every note received on the MIDI input. It generates MIDI notes on csound's MIDI output, so be sure to connect something.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac        -iadc     -d          -M0  -Q1  ;;RT audio I/O with MIDI in
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

; Example by Giorgio Zucco 2007

instr 1 ; Triggered by MIDI notes on channel 1

  ivel veloc

  krate = 8
  iscale = 100 ;f

  ; Random sequence from table f100

```

```
krnd randh int(14),krate,-1
knote table abs(krnd),iscale
; Generates random notes from the scale on ftable 100
; on channel 1 of csound's MIDI output
midion 1,knote,ivel

endin

</CsInstruments>
<CsScore>
f100 0 32 -2 40 50 60 70 80 44 54 65 74 84 39 49 69 69

; Dummy ftable
f0 60
</CsScore>
</CsoundSynthesizer>
```

## See Also

*moscil, midion2, noteon, noteoff, noteondur, noteondur2*

## Credits

Author: Gabriel Maldonado  
Italy  
May 1997

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# midout

midout — Sends a generic MIDI message to the MIDI OUT port.

## Description

Sends a generic MIDI message to the MIDI OUT port.

## Syntax

```
midout kstatus, kchan, kdata1, kdata2
```

## Performance

*kstatus* -- the type of MIDI message. Can be:

- 128 (note off)
- 144 (note on)
- 160 (polyphonic aftertouch)
- 176 (control change)
- 192 (program change)
- 208 (channel aftertouch)
- 224 (pitch bend)
- 0 when no MIDI messages must be sent to the MIDI OUT port

*kchan* -- MIDI channel (1-16)

*kdata1*, *kdata2* -- message-dependent data values

*midout* has no output arguments, because it sends a message to the MIDI OUT port implicitly. It works at k-rate. It sends a MIDI message only when *kstatus* is non-zero.



### Warning

*Warning:* Normally *kstatus* should be set to 0. Only when the user intends to send a MIDI message, can it be set to the corresponding message type number.

## Examples

Here is an example of the midout opcode. It uses the file *midout.csd* [examples/midout.csd].

### Example 551. Example of the midout opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac -Ma -Q1 ;;realtime audio out and midi out and midi in (all midi inputs)
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

midiout 192, 1, 21, 0 ;program change to instr. 21
inum notnum
ivel veloc
midion 1, inum, ivel

endin
</CsInstruments>
<CsScore>

i 1 0 3 80 100 ;play note for 3 seconds

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Gabriel Maldonado  
Italy  
1998

New in Csound version 3.492

# midipitchbend

midipitchbend — Gets a MIDI pitchbend value.

## Description

*midipitchbend* is designed to simplify writing instruments that can be used interchangeably for either score or MIDI input, and to make it easier to adapt instruments originally written for score input to work with MIDI input.

In general, it should be possible to write instrument definitions that work identically with both scores and MIDI, including both MIDI files and real-time MIDI input, without using any conditional statements, and that take full advantage of MIDI voice messages.

Note that correlating Csound instruments with MIDI channel numbers is done using the *massign* opcode for real-time performance. For file-driven performance, instrument numbers default to MIDI channel number + 1, but the defaults are overridden by any MIDI program change messages in the file.

## Syntax

```
midipitchbend xpitchbend [, ilow] [, ihigh]
```

## Initialization

*ilow* (optional) -- optional low value after rescaling, defaults to 0.

*ihigh* (optional) -- optional high value after rescaling, defaults to 127.

## Performance

*xpitchbend* -- returns the MIDI pitch bend during MIDI activation, remains unchanged otherwise.

If the instrument was activated by MIDI input, the opcode overwrites the value of *xpitchbend* with the corresponding value from MIDI input. If the instrument was *NOT* activated by MIDI input, the value of *xpitchbend* remains unchanged.

This enables score p-fields to receive MIDI input data during MIDI activation, and score values otherwise.



### Adapting a score-activated Csound instrument.

See the *MIDI interop opcodes* section for details on adapting score driven instruments for MIDI or vice-versa.

## Examples

Here is an example of the midipitchbend opcode. It uses the file *midipitchbend.csd* [examples/midipitchbend.csd].

### Example 552. Example of the midipitchbend opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -M1 ;;realtime audio out and midi in
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o midipitchbend.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

midinoteonoct p4, p5 ;gets a MIDI note number value as octave-point-decimal value into p4, and MIDI

kvel = p5/127 ;scale midi velocity to 0-1
kpb init 0
midipitchbend kpb
printk2 kpb ;display the pitch-bend value when it changes
koct = p4+kpb ;add pitchbend values to octave-point-decimal value
kcps = cpsoct(koct) ;convert octave-point-decimal value into Hz
kenv madsr 0.5, 0.8, 0.8, 0.5 ;amplitude envelope multiplied by
asig vco2 kenv*kvel, kcps ;velocity value
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 0 30 ;runs for 30 seconds
f 2 0 4096 10 1

i 1 0 2 8.000 100 ; play these notes from score as well
i 1 + 2 8.917 100
e

</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like:

```

i1 0.12695
i1 0.00000
i1 -0.01562

```

## See Also

*midichannelaftertouch*, *midicontrolchange*, *mididefault*, *midinoteoff*, *midinoteoncps*, *midinoteonkey*, *midinoteonoct*, *midinoteonpch*, *midipolyaftertouch*, *midiprogramchange*

## Credits

Author: Michael Gogins

New in version 4.20

# midipolyaftertouch

midipolyaftertouch — Gets a MIDI polyphonic aftertouch value.

## Description

*midipolyaftertouch* is designed to simplify writing instruments that can be used interchangeably for either score or MIDI input, and to make it easier to adapt instruments originally written for score input to work with MIDI input.

In general, it should be possible to write instrument definitions that work identically with both scores and MIDI, including both MIDI files and real-time MIDI input, without using any conditional statements, and that take full advantage of MIDI voice messages.

Note that correlating Csound instruments with MIDI channel numbers is done using the *massign* opcode for real-time performance. For file-driven performance, instrument numbers default to MIDI channel number + 1, but the defaults are overridden by any MIDI program change messages in the file.

## Syntax

```
midipolyaftertouch xpolyaftertouch, xcontrollervalue [, ilow] [, ihigh]
```

## Initialization

*ilow* (optional) -- optional low value after rescaling, defaults to 0.

*ihigh* (optional) -- optional high value after rescaling, defaults to 127.

## Performance

*xpolyaftertouch* -- returns MIDI polyphonic aftertouch during MIDI activation, remains unchanged otherwise.

*xcontrollervalue* -- returns the value of the MIDI controller during MIDI activation, remains unchanged otherwise.

If the instrument was activated by MIDI input, the opcode overwrites the values of *xpolyaftertouch* and *xcontrollervalue* with the corresponding values from MIDI input. If the instrument was *NOT* activated by MIDI input, the values of *xpolyaftertouch* and *xcontrollervalue* remain unchanged.

This enables score p-fields to receive MIDI input data during MIDI activation, and score values otherwise.



### Adapting a score-activated Csound instrument.

See the *MIDI interop opcodes* section for details on adapting score driven instruments for MIDI or vice-versa.

## Examples

Here is an example of the *midipolyaftertouch* opcode. It uses the file *midipolyaftertouch.csd* [examples/midipolyaftertouch.csd].

**Example 553. Example of the midipolyaftertouch opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -M1 ;;realtime audio out and midi in
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o midipolyaftertouch.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1;midi keyboard and score are used

midinoteoncps p4, p5 ;puts MIDI key translated to cycles per second into p4, and MIDI velocity into

kafter init 127 ;full open
midipolyaftertouch kafter, 1
printk2 kafter ;display the key value when it changes and when key is pressed

kvel = p5/127 ;scale midi velocity to 0-1
kenv madsr 0.5, 0.8, 0.8, 0.5 ;amplitude envelope multiplied by
ain pluck kenv*kvel, p4, p4, 2, 1 ;velocity value
asig moogvcf2 ain, 300+(kafter*100), .5 ;use value of aftertouch to control frequency of filter
outs asig, asig ;base freq of filter = 300 Hz

endin
</CsInstruments>
<CsScore>
f 0 30 ;runs for 30 seconds
f 2 0 4096 10 1 ;sine wave

i 1 0 2 440 100 ;play these notes from score as well
i 1 + 2 1440 100
e

</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like:

```

i1 127.00000
i1 20.00000
i1 44.00000

```

## See Also

*midichannelaftertouch*, *midicontrolchange*, *mididefault*, *midinoteoff*, *midinoteoncps*, *midinoteonkey*, *midinoteonoct*, *midinoteonpch*, *midipitchbend*, *midiprogramchange*

## Credits

Author: Michael Gogins

New in version 4.20

# midiprogramchange

midiprogramchange — Gets a MIDI program change value.

## Description

*midiprogramchange* is designed to simplify writing instruments that can be used interchangeably for either score or MIDI input, and to make it easier to adapt instruments originally written for score input to work with MIDI input.

In general, it should be possible to write instrument definitions that work identically with both scores and MIDI, including both MIDI files and real-time MIDI input, without using any conditional statements, and that take full advantage of MIDI voice messages.

Note that correlating Csound instruments with MIDI channel numbers is done using the *massign* opcode for real-time performance. For file-driven performance, instrument numbers default to MIDI channel number + 1, but the defaults are overridden by any MIDI program change messages in the file.

## Syntax

```
midiprogramchange xprogram
```

## Performance

*xprogram* -- returns the MIDI program change value during MIDI activation, remains unchanged otherwise.

If the instrument was activated by MIDI input, the opcode overwrites the value of *xprogram* with the corresponding value from MIDI input. If the instrument was *NOT* activated by MIDI input, the value of *xprogram* remains unchanged.

This enables score p-fields to receive MIDI input data during MIDI activation, and score values otherwise.



### Adapting a score-activated Csound instrument.

See the *MIDI interop opcodes* section for details on adapting score driven instruments for MIDI or vice-versa.

## Examples

Here is an example of the midiprogramchange opcode. It uses the file *midiprogramchange.csd* [examples/midiprogramchange.csd].

### Example 554. Example of the midiprogramchange opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
```

```
-odac -M1 -Q1 ;;;realtime audio out and midi in and midi out
;-iadc ;;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o midiprogramchange.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
Odbfs = 1

instr 1 ;uses external midi device

kkey init 0 ;initialize key number
kvel init 0 ;initialize velocity
midiprogramchange 52 ;choose GM program number 52 = string pad
midinoteonkey kkey, kvel ;MIDI note number value
printk2 kkey ;display the key value when it changes and when key is pressed
midion 1, kkey, kvel ;sent note to external device

endin
</CsInstruments>
<CsScore>
f 0 30 ;runs for 30 seconds

i 1 0 2 62 ; play these notes from score as well
i 1 + 2 65
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*midichannelaftertouch, midicontrolchange, mididefault, midinoteoff, midinoteoncps, midinoteonkey, midinoteonoct, midinoteonpch, midipitchbend, midipolyaftertouch*

## Credits

Author: Michael Gogins

New in version 4.20

# miditempo

miditempo — Returns the current tempo at k-rate, of either the MIDI file (if available) or the score

## Description

Returns the current tempo at k-rate, of either the MIDI file (if available) or the score

## Syntax

```
ksig miditempo
```

## Examples

Here is an example of the miditempo opcode. It uses the files *miditempo.csd* [examples/miditempo.csd].

### Example 555. Example of the miditempo opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -F Anna.mid ;;realtime audio out and midi file input
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o miditempo.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

massign 0, 1 ; make sure that all channels
pgmassign 0, 1 ; and programs are assigned to test instr

instr 1

ksig miditempo
prints "miditempo = %d\\n", ksig

icps cpsmidi ; convert midi note to pitch
kenv madsr 0.1, 0, 0.8, 0.3
asig pluck kenv*.15, icps, icps, 1, 1 ;low volume
outs asig, asig

endin
</CsInstruments>
<CsScore>

f 0 200 ;stay active for 120 seconds
f 1 0 4096 10 1 ;sine

e
</CsScore>
```

```
</CsoundSynthesizer>
```

Its output should include a line like this:

```
miditempo = 96
```

## Credits

Author: Istvan Varga  
March 2005  
New in Csound5

# midremot

midremot — An opcode which can be used to implement a remote midi orchestra. This opcode will send midi events from a source machine to one destination.

## Description

With the *midremot* and *midglobal* opcodes you are able to perform instruments on remote machines and control them from a master machine. The remote opcodes are implemented using the master/client model. All the machines involved contain the same orchestra but only the master machine contains the information of the midi score. During the performance the master machine sends the midi events to the clients. The *midremot* opcode will send events from a source machine to one destination if you want to send events to many destinations (broadcast) use the *midglobal* opcode instead. These two opcodes can be used in combination.

## Syntax

```
midremot idestination, ource, instrnum [,instrnum...]
```

## Initialization

*idestination* -- a string that is the intended host computer (e.g. 192.168.0.100). This is the destination host which receives the events from the given instrument.

*ource* -- a string that is the intended host computer (e.g. 192.168.0.100). This is the source host which generates the events of the given instrument and sends it to the address given by *idestination*.

*instrnum* -- a list of instrument numbers which will be played on the destination machine

## Examples

Here is an example of the midremot opcode. It uses the files *insremot.csd* [examples/midremot.csd].

### Example 556. Example of the insremot opcode.

The example shows a Bach fugue played on 4 remote computers. The master machine is named "192.168.1.100", client1 "192.168.1.101" and so on. Start the clients on each machine (they will be waiting to receive the events from the master machine) and then start the master. Here is the command on linux to start a client (csound -dm0 -odac -+rtaudio=alsa midremot.csd -+rtmidi=None), and the command on the master machine will look like this (csound -dm0 -odac -+rtaudio=alsa midremot.csd -F midremot.mid).

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>

<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o midremot.wav -W ;; for file output any platform
</CsOptions>

<CsInstruments>
```

```

sr = 44100
kr = 441
ksmps = 100
nchnls = 2

massign 1,1
massign 2,2
massign 3,3
massign 4,4
massign 5,5

ga1 init 0
ga2 init 0

gi1 sfloat "19Trumpet.sf2"

gi2 sfloat "01hpschd.sf2"

gi3 sfloat "07AcousticGuitar.sf2"

gi4 sfloat "22Bassoon.sf2"

gitab ftgen 1,0,1024,10,1

midremot "192.168.1.100", "192.168.1.101", 1
midremot "192.168.1.100", "192.168.1.102", 2
midremot "192.168.1.100", "192.168.1.103", 3

midglobal "192.168.1.100", 5

instr 1
  sfpassign 0, gi1
  ifreq cpsmidi
  iamp ampmidi 10
  inum notnum
  ivel veloc
  kamp linsegr 1,1,1,.1,0
  kfreq init 1
  a1,a2 sfloat ivel,inum,kamp*iamp,kfreq,0,0
  outs a1,a2
  vincr ga1, a1*.5
  vincr ga2, a2*.5
  endin

instr 2
  sfpassign 0, gi2
  ifreq cpsmidi
  iamp ampmidi 15
  inum notnum
  ivel veloc
  kamp linsegr 1,1,1,.1,0
  kfreq init 1
  a1,a2 sfloat ivel,inum,kamp*iamp,kfreq,0,0
  outs a1,a2
  vincr ga1, a1*.4
  vincr ga2, a2*.4
  endin

instr 3
  sfpassign 0, gi3
  ifreq cpsmidi
  iamp ampmidi 10
  inum notnum
  ivel veloc
  kamp linsegr 1,1,1,.1,0
  kfreq init 1

```

```
a1,a2 sfplay ivel,inum,kamp*iamp,kfreq,0,0
outs a1,a2
vincr ga1, a1*.5
vincr ga2, a2*.5
endin

instr 4
sfpassign 0, gi4
ifreq cpsmidi
iamp ampmidi 15
inum notnum
ivel veloc
kamp linsegr 1,1,1,.1,0
kfreq init 1
a1,a2 sfplay ivel,inum,kamp*iamp,kfreq,0,0
outs a1,a2
vincr ga1, a1*.5
vincr ga2, a2*.5
endin

instr 5
kamp midic7 1,0,1
denorm ga1
denorm ga2
aL, aR reverbsc ga1, ga2, .9, 16000, sr, 0.5
outs aL, aR
ga1 = 0
ga2 = 0
endin

</CsInstruments>
<CsScore>
; Score
f0 160
</CsScore>

</CsoundSynthesizer>
```

## See also

*insglobal, insremot, midglobal, remoteport*

## Credits

Author: Simon Schampijer  
2006

New in version 5.03

# min

min — Produces a signal that is the minimum of any number of input signals.

## Description

The *min* opcode takes any number of a-rate, k-rate or i-rate signals as input (all of the same rate), and outputs a signal at the same rate that is the minimum of all of the inputs. For a-rate signals, the inputs are compared one sample at a time (i.e. *min* does not scan an entire ksmps period of a signal for its local minimum as the *max\_k* opcode does).

## Syntax

```
amin min ain1, ain2 [, ain3] [, ain4] [...]
```

```
kmin min kin1, kin2 [, kin3] [, kin4] [...]
```

```
imin min iin1, iin2 [, iin3] [, iin4] [...]
```

## Performance

*ain1, ain2, ...* -- a-rate signals to be compared.

*kin1, kin2, ...* -- k-rate signals to be compared.

*iin1, iin2, ...* -- i-rate signals to be compared.

## Examples

Here is an example of the min opcode. It uses the file *min.csd* [examples/min.csd].

### Example 557. Example of the min opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;;realtime audio out
;-iadc   ;;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o min.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

k1  oscili 1, 10.0, 1   ;combine 3 sinusses
k2  oscili 1, 1.0, 1   ;at different rates
```

```
k3 oscili 1, 3.0, 1
kmin min k1, k2, k3
kmin = kmin*250 ;scale kmin
printk2 kmin ;check the values

aout vco2 .5, 220, 6 ;sawtooth
asig moogvcf2 aout, 600+kmin, .5 ;change filter around 600 Hz
outs asig, asig

endin

</CsInstruments>
<CsScore>

f1 0 32768 10 1

i1 0 5
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*max, maxabs, minabs, maxaccum, minaccum, maxabsaccum, minabsaccum, max\_k*

## Credits

Author: Anthony Kozar  
March 2006

New in Csound version 5.01; i-rate version new in 6.04

# minabs

minabs — Produces a signal that is the minimum of the absolute values of any number of input signals.

## Description

The *minabs* opcode takes any number of a-rate or k-rate signals as input (all of the same rate), and outputs a signal at the same rate that is the minimum of all of the inputs. It is identical to the *min* opcode except that it takes the absolute value of each input before comparing them. Therefore, the output is always non-negative. For a-rate signals, the inputs are compared one sample at a time (i.e. *minabs* does not scan an entire ksmps period of a signal for its local minimum as the *max\_k* opcode does).

## Syntax

```
amin minabs ain1, ain2 [, ain3] [, ain4] [...]  
kmin minabs kin1, kin2 [, kin3] [, kin4] [...]
```

## Performance

*ain1, ain2, ...* -- a-rate signals to be compared.

*kin1, kin2, ...* -- k-rate signals to be compared.

## Examples

Here is an example of the minabs opcode. It uses the file *minabs.csd* [examples/minabs.csd].

### Example 558. Example of the minabs opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
-odac   ;;realtime audio out  
;-iadc   ;;uncomment -iadc if realtime audio input is needed too  
; For Non-realtime ouput leave only the line below:  
; -o minabs.wav -W   ;; for file output any platform  
</CsOptions>  
<CsInstruments>  
  
sr = 44100  
ksmps = 32  
nchnls = 2  
0dbfs = 1  
  
instr 1  
  
k1  oscili 1, 10.0, 1   ;combine 3 sinusses  
k2  oscili 1, 1.0, 1   ;at different rates  
k3  oscili 1, 3.0, 1  
kmin minabs k1, k2, k3  
kmin = kmin*250   ;scale kmin
```

```
printk2 kmin      ;check the values

aout vco2 .5, 220, 6 ;sawtooth
asig moogvcf2 aout, 600+kmin, .5 ;change filter above 600 Hz
outs asig, asig

endin

</CsInstruments>
<CsScore>
f1 0 32768 10 1

i1 0 5
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*maxabs, max, min, maxaccum, minaccum, maxabsaccum, minabsaccum, max\_k*

## Credits

Author: Anthony Kozar  
March 2006

New in Csound version 5.01

# minabsaccum

minabsaccum — Accumulates the minimum of the absolute values of audio signals.

## Description

*minabsaccum* compares two audio-rate variables and stores the minimum of their absolute values into the first.

## Syntax

```
minabsaccum aAccumulator, aInput
```

## Performance

*aAccumulator* -- audio variable to store the minimum value

*aInput* -- signal that *aAccumulator* is compared to

The *minabsaccum* opcode is designed to accumulate the minimum value from among many audio signals that may be in different note instances, different channels, or otherwise cannot all be compared at once using the *minabs* opcode. *minabsaccum* is identical to *minaccum* except that it takes the absolute value of *aInput* before the comparison. Its semantics are similar to *vincr* since *aAccumulator* is used as both an input and an output variable, except that *minabsaccum* keeps the minimum absolute value instead of adding the signals together. *minabsaccum* performs the following operation on each pair of samples:

$$\text{if } (\text{abs}(\text{aInput}) < \text{aAccumulator}) \text{ aAccumulator} = \text{abs}(\text{aInput})$$

*aAccumulator* will usually be a global audio variable. At the end of any given computation cycle (k-period), after its value is read and used in some way, the accumulator variable should usually be reset to some large enough positive value that will always be greater than the input signals to which it is compared.

## Examples

Here is an example of the *minabsaccum* opcode. It uses the file *minabsaccum.csd* [examples/minabsaccum.csd].

### Example 559. Example of the minabsaccum opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o minabsaccum.wav -W ;; for file output any platform
</CsOptions>
```

```
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;saw

asig vco2 .2, p4
outs asig, asig
gasaw = asig
endin

instr 2 ;sine

aout poscil .3, p4, 1
outs aout, aout
gasin = aout
endin

instr 10

accum init 0
minabsaccum accum, gasaw + gasin ;saw and sine accumulated
accum dcblock2 accum ;get rid of DC
outs accum, accum

clear accum
endin

</CsInstruments>
<CsScore>
f 1 0 4096 10 1

i 1 0 7 330
i 2 3 3 440

i 1 10 7 330 ;same notes but without minabsaccum, for comparison
i 2 13 3 440

i 10 0 6 ;accumulation note stops after 6 seconds

e
</CsScore>
</CsSoundSynthesizer>
```

## See Also

*maxabsaccum, maxaccum, minaccum, max, min, maxabs, minabs, vincr*

## Credits

Author: Anthony Kozar  
March 2006

New in Csound version 5.01

# minaccum

`minaccum` — Accumulates the minimum value of audio signals.

## Description

`minaccum` compares two audio-rate variables and stores the minimum value between them into the first.

## Syntax

```
minaccum aAccumulator, aInput
```

## Performance

*aAccumulator* -- audio variable to store the minimum value

*aInput* -- signal that *aAccumulator* is compared to

The `minaccum` opcode is designed to accumulate the minimum value from among many audio signals that may be in different note instances, different channels, or otherwise cannot all be compared at once using the `min` opcode. Its semantics are similar to `vincr` since *aAccumulator* is used as both an input and an output variable, except that `minaccum` keeps the minimum value instead of adding the signals together. `minaccum` performs the following operation on each pair of samples:

```
if (aInput < aAccumulator) aAccumulator = aInput
```

*aAccumulator* will usually be a global audio variable. At the end of any given computation cycle (k-period), after its value is read and used in some way, the accumulator variable should usually be reset to some large enough positive value that will always be greater than the input signals to which it is compared.

## Examples

Here is an example of the `minaccum` opcode. It uses the file `minaccum.csd` [examples/minaccum.csd].

### Example 560. Example of the `minaccum` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o minaccum.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
```

```
nchnls = 2
0dbfs = 1

instr 1 ;saw

asig vco2 .2, p4
outs asig, asig
gasaw = asig
endin

instr 2 ;sine

aout poscil .3, p4, 1
outs aout, aout
gasin = aout
endin

instr 10

accum init 0
minaccum accum, gasaw + gasin ;saw and sine accumulated
outs accum, accum

clear accum
endin

</CsInstruments>
<CsScore>
f 1 0 4096 10 1

i 1 0 7 330
i 2 3 3 440

i 1 10 7 330 ;same notes but without minaccum, for comparison
i 2 13 3 440

i 10 0 6 ;accumulation note stops after 6 seconds

e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*maxaccum, maxabsaccum, minabsaccum, max, min, maxabs, minabs, vincr*

## Credits

Author: Anthony Kozar  
March 2006

New in Csound version 5.01

# mincer

mincer — Phase-locked vocoder processing.

## Description

*mincer* implements phase-locked vocoder processing using function tables containing sampled-sound sources, with *GEN01*, and *mincer* will accept deferred allocation tables.

This opcode allows for time and frequency-independent scaling. Time is controlled by a time index (in seconds) to the function table position and can be moved forward and backward at any chosen speed, as well as stopped at a given position ("frozen"). The quality of the effect is generally improved with phase locking switched on.

*mincer* will also scale pitch, independently of frequency, using a transposition factor (k-rate).

## Syntax

```
asig mincer atimpt, kamp, kpitch, ktab, klock[,ifftsize,idecim]
```

## Initialization

*ifftsize* -- FFT size (power-of-two), defaults to 2048.

*idecim* -- decimation, defaults to 4 (meaning hopsize = fftsize/4)

## Performance

*atimpt* -- time position of current audio sample in secs. Table reading wraps around the ends of the function table.

*kamp* -- amplitude scaling

*kpitch* -- grain pitch scaling (1=normal pitch, < 1 lower, > 1 higher; negative, backwards)

*klock* -- 0 or 1, to switch phase-locking on/off

*ktab* -- source signal function table. Deferred-allocation tables (see *GEN01*) are accepted, but the opcode expects a mono source. Tables can be switched at k-rate.

## Examples

Here is an example of the *mincer* opcode. It uses the file *mincer.csd* [examples/mincer.csd]

### Example 561. Example of the mincer opcode.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
-odac    ;;;realtime audio out
```

```
;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o mincer.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

idur = p3
ilock = p4
ipitch = 1
itimescale = 0.5
iamp = 0.8

atime line 0,idur,idur*itimescale
asig mincer atime, iamp, ipitch, 1, ilock
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 0 1 "fox.wav" 0 0 0

i 1 0 5 0 ;not locked
i 1 6 5 1 ;locked

e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
February 2010

New plugin in version 5.13

February 2005.

# minarray

minarray — returns the minimum value in an array.

## Description

The *minarray* opcode returns the minimum value in a k-rate array, and optional its index.

## Syntax

```
kmin [,kindx] minarray karray
```

## Performance

*kmin* -- variable for result.

*kindx* -- position (index) of result in array.

*karray* -- array for reading.

## Examples

Here is an example of the minarray opcode. It uses the file *minarray.csd* [examples/minarray.csd].

### Example 562. Example of the minarray opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n
</CsOptions>
<CsInstruments>
ksmps = 32
;example by joachim heintz

          seed          0

instr 1
;create an array with 10 elements
kArr[] init          10
;fill in random numbers and print them out
kIndx = 0
  until kIndx == 10 do
kNum random          -100, 100
kArr[kIndx] = kNum
printf             "kArr[%d] = %10f\n", kIndx+1, kIndx, kNum
kIndx +=           1
  od
;investigate minimum number and print it out
kMin, kMinIndx minarray kArr
printf             "Minimum of kArr = %f at index %d\n", kIndx+1, kMin, kMinIndx
turnoff
endin
```

```
</CsInstruments>  
<CsScore>  
i1 0 0.1  
e  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*maxarray*, *sumarray*, *scalearray*,

## Credits

Author: John fitch  
October 2011

New in Csound version 5.14

Revised in Csound version 6.00 to work on multidimensional arrays

# mirror

mirror — Reflects the signal that exceeds the low and high thresholds.

## Description

Reflects the signal that exceeds the low and high thresholds.

## Syntax

```
ares mirror asig, klow, khigh
ires mirror isig, ilow, ihigh
kres mirror ksig, klow, khigh
```

## Initialization

```
isig -- input signal
ilow -- low threshold
ihigh -- high threshold
```

## Performance

```
xsig -- input signal
klow -- low threshold
khigh -- high threshold
mirror “reflects” the signal that exceeds the low and high thresholds.
```

This opcode is useful in several situations, such as table indexing or for clipping and modeling a-rate, i-rate or k-rate signals.

## Examples

Here is an example of the mirror opcode. It uses the file *mirror.csd* [examples/mirror.csd].

### Example 563. Example of the mirror opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o mirror.wav -W ;; for file output any platform
```

```

</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
Odbfs = 1
nchnls = 2

instr 1 ; Limit / Mirror / Wrap

igain = p4 ;gain
ilevl1 = p5 ; + level
ilevl2 = p6 ; - level
imode = p7 ;1 = limit, 2 = mirror, 3 = wrap

ain soundin "fox.wav"
ain = ain*igain

if imode = 1 goto limit
if imode = 2 goto mirror

asig wrap ain, ilevl2, ilevl1
goto outsignal

limit:
asig limit ain, ilevl2, ilevl1
goto outsignal

mirror:
asig mirror ain, ilevl2, ilevl1
outsignal:

outs asig*.5, asig*.5 ;mind your speakers

endin

</CsInstruments>
<CsScore>

;          Gain +Levl -Levl Mode
i1 0 3 4.00 .25 -1.00 1 ;limit
i1 4 3 4.00 .25 -1.00 2 ;mirror
i1 8 3 4.00 .25 -1.00 3 ;wrap
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*limit, wrap*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.49

# MixerSetLevel

MixerSetLevel — Sets the level of a send to a buss.

## Syntax

```
MixerSetLevel isend, ibuss, kgain
```

## Description

Sets the level at which signals from the send are added to the buss. The actual sending of the signal to the buss is performed by the *MixerSend* opcode.

## Initialization

*isend* -- The number of the send, for example the number of the instrument sending the signal (but any integer can be used).

*ibuss* -- The number of the buss, for example the number of the instrument receiving the signal (but any integer can be used).

Setting the gain for a buss also creates the buss.

## Performance

*kgain* -- The level (any real number) at which the signal from the send will be mixed onto the buss. The default is 0.

Use of the mixer requires that instruments setting gains have smaller numbers than instruments sending signals, and that instruments sending signals have smaller numbers than instruments receiving those signals. However, an instrument may have any number of sends or receives. After the final signal is received, *MixerClear* must be invoked to reset the busses before the next kperiod.

## Examples

In the orchestra, define an instrument to control mixer levels:

```
instr 1
  MixerSetLevel p4, p5, p6
endin
```

In the score, use that instrument to set mixer levels:

```
; SoundFonts
; to Chorus
i 1 0 0 100 200 0.9
; to Reverb
i 1 0 0 100 210 0.7
; to Output
i 1 0 0 100 220 0.3

; Kelley Harpsichord
```

```

; to Chorus
i 1 0 0 3 200 0.30
; to Reverb
i 1 0 0 3 210 0.9
; to Output
i 1 0 0 3 220 0.1

; Chorus to Reverb
i 1 0 0 200 210 0.5
; Chorus to Output
i 1 0 0 200 220 0.5
; Reverb to Output
i 1 0 0 210 220 0.2

```

Here is a complete example of the MixerSetLevel opcode. It uses the file *Mixer.csd* [examples/Mixer.csd]

### Example 564. Complete example of the MixerSetLevel opcode.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      ; -iadc      ;;RT audio out
; For Non-realtime output leave only the line below:
; -o Mixer.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

katt expon 0.01, p3, 1 ;create an attack
aout poscil .7, 440,1
  MixerSetLevel 1, 3, katt ;impose attack on the gain level
  MixerSend aout, 1, 3, 0 ;send to channel 0
endin

instr 2

aout vco2 .5, 110 ;saw wave
  MixerSetLevel 2, 3, .25 ;set level to .25 of vco2
  MixerSend aout, 2, 3, 1 ;send to channel 1
endin

instr 3 ;mix instr.1 and 2 with reverb

kgain1 MixerGetLevel 1,3 ;get level form buss 3
kgain2 MixerGetLevel 2,3 ;get level form buss 3
a1 MixerReceive 3,0 ;receive channel 0
a2 MixerReceive 3,1 ;receive channel 1
aout = a1*kgain1+a2*kgain2 ;mix them
aoutL, aoutR reverbsc aout, aout, 0.85, 12000 ;add a nice reverb
outs aoutL, aoutR
  MixerClear
endin

</CsInstruments>
<CsScore>
f1 0 4096 10 1

i1 0 2

```

```
i2 0 2  
i3 0 8 ;reverb stays on for 8 sec.  
  
e  
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Michael Gogins (gogins at pipeline dot com).

# MixerSetLevel\_i

MixerSetLevel\_i — Sets the level of a send to a buss.

## Syntax

```
MixerSetLevel_i isend, ibuss, igain
```

## Description

Sets the level at which signals from the send are added to the buss. This opcode, because all parameters are irate, may be used in the orchestra header. The actual sending of the signal to the buss is performed by the *MixerSend* opcode.

## Initialization

*isend* -- The number of the send, for example the number of the instrument sending the signal (but any integer can be used).

*ibuss* -- The number of the buss, for example the number of the instrument receiving the signal (but any integer can be used).

*igain* -- The level (any real number) at which the signal from the send will be mixed onto the buss. The default is 0.

Setting the gain for a buss also creates the buss.

## Performance

Use of the mixer requires that instruments setting gains have smaller numbers than instruments sending signals, and that instruments sending signals have smaller numbers than instruments receiving those signals. However, an instrument may have any number of sends or receives. After the final signal is received, *MixerClear* must be invoked to reset the busses before the next kperiod.

## Examples

In the orchestra header, set the gain for the send from buss 3 to buss 4:

```
MixerSetLevel_i 3, 4, 0.76
```

## Credits

Michael Gogins (gogins at pipeline dot com).

# MixerGetLevel

MixerGetLevel — Gets the level of a send to a buss.

## Syntax

```
kgain MixerGetLevel isend, ibuss
```

## Description

Gets the level at which signals from the send are being added to the buss. The actual sending of the signal to the buss is performed by the *MixerSend* opcode.

## Initialization

*isend* -- The number of the send, for example the number of the instrument sending the signal.

*ibuss* -- The number of the buss, for example the number of the instrument receiving the signal.

## Performance

*kgain* -- The level (any real number) at which the signal from the send will be mixed onto the buss.

This opcode reports the level set by *MixerSetLevel* for a send and buss pair.

Use of the mixer requires that instruments setting gains have smaller numbers than instruments sending signals, and that instruments sending signals have smaller numbers than instruments receiving those signals. However, an instrument may have any number of sends or receives. After the final signal is received, *MixerClear* must be invoked to reset the busses to 0 before the next kperiod.

## Examples

Here is an example of the MixerGetLevel opcode. It uses the file *Mixer.csd* [examples/Mixer.csd]

### Example 565. Example of the MixerGetLevel opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      ; -iadc      ;;RT audio out
; For Non-realtime ouput leave only the line below:
; -o Mixer.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
```

```
katt expon 0.01, p3, 1 ;create an attack
aout poscil .7, 440,1
  MixerSetLevel 1, 3, katt ;impose attack on the gain level
  MixerSend aout, 1, 3, 0 ;send to channel 0
endin

instr 2

aout vco2 .5, 110 ;saw wave
  MixerSetLevel 2, 3, .25 ;set level to .25 of vco2
  MixerSend aout, 2, 3, 1 ;send to channel 1
endin

instr 3 ;mix instr.1 and 2 with reverb

kgain1 MixerGetLevel 1,3 ;get level form buss 3
kgain2 MixerGetLevel 2,3 ;get level form buss 3
a1 MixerReceive 3,0 ;receive channel 0
a2 MixerReceive 3,1 ;receive channel 1
aout = a1*kgain1+a2*kgain2 ;mix them
aoutL, aoutR reverbsc aout, aout, 0.85, 12000 ;add a nice reverb
  outs aoutL, aoutR
  MixerClear
endin

</CsInstruments>
<CsScore>
f1 0 4096 10 1

i1 0 2
i2 0 2
i3 0 8 ;reverb stays on for 8 sec.

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Michael Gogins (gogins at pipeline dot com).

# MixerSend

MixerSend — Mixes an arate signal into a channel of a buss.

## Syntax

```
MixerSend asignal, isend, ibuss, ichannel
```

## Description

Mixes an arate signal into a channel of a buss.

## Initialization

*isend* -- The number of the send, for example the number of the instrument sending the signal. The gain of the send is controlled by the *MixerSetLevel* opcode. The reason that the sends are numbered is to enable different levels for different sends to be set independently of the actual level of the signals.

*ibuss* -- The number of the buss, for example the number of the instrument receiving the signal.

*ichannel* -- The number of the channel. Each buss has `nchnls` channels.

## Performance

*asignal* -- The signal that will be mixed into the indicated channel of the buss.

Use of the mixer requires that instruments setting gains have smaller numbers than instruments sending signals, and that instruments sending signals have smaller numbers than instruments receiving those signals. However, an instrument may have any number of sends or receives. After the final signal is received, *MixerClear* must be invoked to reset the busses to 0 before the next kperiod.

## Examples

```
instr 100 ; Fluidsynth output
; INITIALIZATION
; Normalize so iamplitude for p5 of 80 == ampdb(80).
iamplitude = ampdb(p5) * 2.0
; AUDIO
aleft, aright fluidAllOut giFluidsynth
asig1 = aleft * iamplitude
asig2 = aright * iamplitude
; To the chorus.
MixerSend asig1, 100, 200, 0
MixerSend asig2, 100, 200, 1
; To the reverb.
MixerSend asig1, 100, 210, 0
MixerSend asig2, 100, 210, 1
; To the output.
MixerSend asig1, 100, 220, 0
MixerSend asig2, 100, 220, 1
endin
```

Here is a complete example of the MixerSend opcode. It uses the file *Mixer.csd* [examples/Mixer.csd]

**Example 566. Complete example of the MixerSend opcode.**

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in
-odac        ; -iadc      ;;RT audio out
; For Non-realtime ouput leave only the line below:
; -o Mixer.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

katt expon 0.01, p3, 1 ;create an attack
aout poscil .7, 440,1
  MixerSetLevel 1, 3, katt ;impose attack on the gain level
  MixerSend aout, 1, 3, 0 ;send to channel 0
endin

instr 2

aout vco2 .5, 110 ;saw wave
  MixerSetLevel 2, 3, .25 ;set level to .25 of vco2
  MixerSend aout, 2, 3, 1 ;send to channel 1
endin

instr 3 ;mix instr.1 and 2 with reverb

kgain1 MixerGetLevel 1,3 ;get level form buss 3
kgain2 MixerGetLevel 2,3 ;get level form buss 3
a1 MixerReceive 3,0 ;receive channel 0
a2 MixerReceive 3,1 ;receive channel 1
aout = a1*kgain1+a2*kgain2 ;mix them
aoutL, aoutR reverbbsc aout, aout, 0.85, 12000 ;add a nice reverb
outs aoutL, aoutR
  MixerClear
endin

</CsInstruments>
<CsScore>
f1 0 4096 10 1

i1 0 2
i2 0 2
i3 0 8 ;reverb stays on for 8 sec.

e
</CsScore>
</CsoundSynthesizer>

```

## Credits

Michael Gogins (gogins at pipeline dot com).

# MixerReceive

MixerReceive — Receives an arate signal from a channel of a buss.

## Syntax

```
asignal MixerReceive ibuss, ichannel
```

## Description

Receives an arate signal that has been mixed onto a channel of a buss.

## Initialization

*ibuss* -- The number of the buss, for example the number of the instrument receiving the signal.

*ichannel* -- The number of the channel. Each buss has `nchnls` channels.

## Performance

*asignal* -- The signal that has been mixed onto the indicated channel of the buss.

Use of the mixer requires that instruments setting gains have smaller numbers than instruments sending signals, and that instruments sending signals have smaller numbers than instruments receiving those signals. However, an instrument may have any number of sends or receives. After the final signal is received, *MixerClear* must be invoked to reset the busses to 0 before the next `kperiod`.

## Examples

```
instr 220 ; Master output
; It applies a bass enhancement, compression and fadeout
; to the whole piece, outputs signals, and clears the mixer.
a1 MixerReceive 220, 0
a2 MixerReceive 220, 1
; Bass enhancement
a11 butterlp a1, 100
a12 butterlp a2, 100
a1 = a11*1.5 + a1
a2 = a12*1.5 + a2

; Global amplitude shape
kenv linseg 0., p5 / 2.0, p4, p3 - p5, p4, p5 / 2.0, 0.
a1=a1*kenv
a2=a2*kenv

; Compression
a1 dam a1, 5000, 0.5, 1, 0.2, 0.1
a2 dam a2, 5000, 0.5, 1, 0.2, 0.1

; Remove DC bias
a1blocked dcblock a1
a2blocked dcblock a2

; Output signals
outs a1blocked, a2blocked
MixerClear
```

endin

Here is a complete example of the MixerReceive opcode. It uses the file *Mixer.csd* [examples/Mixer.csd]

### Example 567. Complete example of the MixerReceive opcode.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      ; -iadc      ;;RT audio out
; For Non-realtime ouput leave only the line below:
; -o Mixer.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

katt expon 0.01, p3, 1 ;create an attack
aout poscil .7, 440,1
  MixerSetLevel 1, 3, katt ;impose attack on the gain level
  MixerSend aout, 1, 3, 0 ;send to channel 0
endin

instr 2

aout vco2 .5, 110 ;saw wave
  MixerSetLevel 2, 3, .25 ;set level to .25 of vco2
  MixerSend aout, 2, 3, 1 ;send to channel 1
endin

instr 3 ;mix instr.1 and 2 with reverb

kgain1 MixerGetLevel 1,3 ;get level form buss 3
kgain2 MixerGetLevel 2,3 ;get level form buss 3
a1 MixerReceive 3,0 ;receive channel 0
a2 MixerReceive 3,1 ;receive channel 1
aout = a1*kgain1+a2*kgain2 ;mix them
aoutL, aoutR reverbbsc aout, aout, 0.85, 12000 ;add a nice reverb
outs aoutL, aoutR
  MixerClear
endin

</CsInstruments>
<CsScore>
f1 0 4096 10 1

i1 0 2
i2 0 2
i3 0 8 ;reverb stays on for 8 sec.

e
</CsScore>
</CsoundSynthesizer>

```

## Credits

Michael Gogins (gogins at pipeline dot com).

# MixerClear

MixerClear — Resets all channels of a buss to 0.

## Syntax

`MixerClear`

## Description

Resets all channels of a buss to 0.

## Performance

Use of the mixer requires that instruments setting gains have smaller numbers than instruments sending signals, and that instruments sending signals have smaller numbers than instruments receiving those signals. However, an instrument may have any number of sends or receives. After the final signal is received, *MixerClear* must be invoked to reset the busses to 0 before the next kperiod.

## Examples

```
instr 220 ; Master output
    ; It applies a bass enhancement, compression and fadeout
    ; to the whole piece, outputs signals, and clears the mixer.
a1 MixerReceive 220, 0
a2 MixerReceive 220, 1
    ; Bass enhancement
a11 butterlp a1, 100
a12 butterlp a2, 100
a1 = a11*1.5 + a1
a2 = a12*1.5 + a2

    ; Global amplitude shape
kenv linseg 0., p5 / 2.0, p4, p3 - p5, p4, p5 / 2.0, 0.
a1=a1*kenv
a2=a2*kenv

    ; Compression
a1 dam a1, 5000, 0.5, 1, 0.2, 0.1
a2 dam a2, 5000, 0.5, 1, 0.2, 0.1

    ; Remove DC bias
a1blocked dcblock a1
a2blocked dcblock a2

    ; Output signals
outs a1blocked, a2blocked
MixerClear
endin
```

Here is a complete example of the Mixerclear opcode. It uses the file *Mixer.csd* [examples/Mixer.csd]

**Example 568. Complete example of the Mixerclear opcode.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      ; -iadc      ;;RT audio out
; For Non-realtime ouput leave only the line below:
; -o Mixer.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

katt expon 0.01, p3, 1 ;create an attack
aout poscil .7, 440,1
  MixerSetLevel 1, 3, katt ;impose attack on the gain level
  MixerSend aout, 1, 3, 0 ;send to channel 0
endin

instr 2

aout vco2 .5, 110 ;saw wave
  MixerSetLevel 2, 3, .25 ;set level to .25 of vco2
  MixerSend aout, 2, 3, 1 ;send to channel 1
endin

instr 3 ;mix instr.1 and 2 with reverb

kgain1 MixerGetLevel 1,3 ;get level form buss 3
kgain2 MixerGetLevel 2,3 ;get level form buss 3
a1 MixerReceive 3,0 ;receive channel 0
a2 MixerReceive 3,1 ;receive channel 1
aout = a1*kgain1+a2*kgain2 ;mix them
aoutL, aoutR reverbbsc aout, aout, 0.85, 12000 ;add a nice reverb
  outs aoutL, aoutR
  MixerClear
endin

</CsInstruments>
<CsScore>
f1 0 4096 10 1

i1 0 2
i2 0 2
i3 0 8 ;reverb stays on for 8 sec.

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Michael Gogins (gogins at pipeline dot com).

# mode

mode — A filter that simulates a mass-spring-damper system

## Description

Filters the incoming signal with the specified resonance frequency and quality factor. It can also be seen as a signal generator for high quality factor, with an impulse for the excitation. You can combine several modes to built complex instruments such as bells or guitar tables.

## Syntax

```
aout mode ain, xfreq, xQ [, iskip]
```

## Initialization

*iskip* (optional, default=0) -- if non zero skip the initialisation of the filter.

## Performance

*aout* -- filtered signal

*ain* -- signal to filter

*xfreq* -- resonant frequency of the filter



### Warning

This filter becomes unstable if  $sr/xfreq < \pi$  (e.g  $xfreq > 14037$  Hz @ 44 kHz)

*xQ* -- quality factor of the filter

The resonance time is roughly proportional to  $xQ/xfreq$ .

See *Modal Frequency Ratios* for frequency ratios of real instruments which can be used to determine the values of *xfreq*.

## Examples

Here is an example of the mode opcode. It uses the file *mode.csd* [examples/mode.csd].

### Example 569. Example of the mode opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc    -d      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o moogvcf.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

instr 1; 2 modes excitator

idur init p3
ifreq11 init p4
ifreq12 init p5
iQ11    init p6
iQ12    init p7
iamp    init ampdb(p8)
ifreq21 init p9
ifreq22 init p10
iQ21    init p11
iQ22    init p12

; to simulate the shock between the excitator and the resonator
ashock mpulse 3,0

aexc1 mode ashock,ifreq11,iQ11
aexc1 = aexc1*iamp
aexc2 mode ashock,ifreq12,iQ12
aexc2 = aexc2*iamp

aexc = (aexc1+aexc2)/2

;"Contact" condition : when aexc reaches 0, the excitator looses
;contact with the resonator, and stops "pushing it"
aexc limit aexc,0,3*iamp

; 2modes resonator

ares1 mode aexc,ifreq21,iQ21
ares2 mode aexc,ifreq22,iQ22

ares = (ares1+ares2)/2

display aexc+ares,p3
outs aexc+ares,aexc+ares

endin

</CsInstruments>
<CsScore>

;wooden excitator against glass resonator
i1 0 8 1000 3000 12 8 70 440 888 500 420

;felt against glass
i1 4 8 80 188 8 3 70 440 888 500 420

;wood against wood
i1 8 8 1000 3000 12 8 70 440 630 60 53

;felt against wood
i1 12 8 80 180 8 3 70 440 630 60 53

i1 16 8 1000 3000 12 8 70 440 888 2000 1630
i1 23 8 80 180 8 3 70 440 888 2000 1630

```

```
;With a metallic excitator  
  
i1 33 8 1000 1800 1000 720 70 440 882 500 500  
i1 37 8 1000 1800 1000 850 70 440 630 60 53  
  
i1 42 8 1000 1800 2000 1720 70 440 442 500 500  
  
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Original UDO and documentation/example by François Blanc

Opcode translation to C-code by Steven Yi

New in version 5.04

Audio rate parameters introduced in version 6.02

November 2013.

# modmatrix

modmatrix — Modulation matrix opcode with optimizations for sparse matrices.

## Description

The opcode can be used to let a large number of k-rate modulator variables modulate a large number of k-rate parameter variables, with arbitrary scaling of each modulator-to-parameter connection. Csound ftables are used to hold both the input (parameter) variables, the modulator variables, and the scaling coefficients. Output variables are written to another Csound ftable.

## Syntax

```
modmatrix iresfn, isrcmodfn, isrcparmfn, imodscale, inum_mod, \\  
inum_parm, kupdate
```

## Initialization

*iresfn* -- ftable number for the parameter output variables

*isrcmodfn* -- ftable number for the modulation source variables

*isrcparmfn* -- ftable number for the parameter input variables

*imodscale* -- scaling/routing coefficient matrix. This is also a csound ftable, used as a matrix of *inum\_mod* rows and *inum\_parm* columns.

*inum\_mod* -- number of modulation variables

*inum\_parm* -- number of parameter (input and output) variables.

The arguments *inum\_mod* and *inum\_parm* do not have to be set to power-of-two values.

## Performance

*kupdate* -- flag to update the scaling coefficients. When the flag is set to a nonzero value, the scaling coefficients are read directly from the *imodscale* ftable. When the flag is set to zero, the scaling coefficients are scanned, and an optimized scaling matrix stored internally in the opcode.

For each modulator in *isrcmodfn*, scale it with the coefficient (in *imodscale*) determining to what degree it should influence each parameter. Then sum all modulators for each parameter and add the resulting modulator value to the input parameter value read from *isrcparmfn*. Finally, write the output parameter values to table *iresfn*.

The following tables give insight into the processing performed by the modmatrix opcode, for a simplified example using 3 parameter and 2 modulators. Let's call the parameters "cps1", "cps2", and "cutoff", and the modulators "lfo1" and "lfo2".

The input variables may at a given point in time have these values:

**Table 12.**

	<b>cps1</b>	<b>cps2</b>	<b>cutoff</b>
<i>isrcparmfn</i>	400	800	3

... while the modulator variables have these values:

**Table 13.**

	<b>lfo1</b>	<b>lfo2</b>
<i>isrcmodfn</i>	0.5	-0.2

The scaling/routing coefficients used:

**Table 14.**

<i>imodscale</i>	<b>cps1</b>	<b>cps2</b>	<b>cutoff</b>
<i>lfo1</i>	40	0	-2
<i>lfo2</i>	-50	100	3

... and the resulting output values:

**Table 15.**

	<b>cps1</b>	<b>cps2</b>	<b>cutoff</b>
<i>iresfn</i>	430	780	1.4
<i>lfo2</i>	-50	100	3

The output value for "cps1" is calculated as  $400+(0.5*40)+(-0.2*-50)$ , similarly for "cps2"  $800+(0.5*0)+(-0.2*100)$ , and for cutoff:  $3+(0.5*-2)+(-0.2*3)$

The imodscale ftable may be specified in the score like this:

```
f1 0 8 -2 200 0 2 50 300 -1.5
```

Or more conveniently using *ftgen* in the orchestra:

```
gimodscale ftgen 0, 0, 8, -2, 200, 0, 2, 50, 300, -1.5
```

Obviously, the parameter and modulator variables need not be static values, and similarly, the scaling routing coefficient table may be continuously rewritten using opcodes like *tablew*.

## Examples

Here is an example of the modmatrix opcode. It uses the file *modmatrix.csd* [examples/modmatrix.csd].

### Example 570. Example of the modmatrix opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio flags here according to platform
; Audio out   Audio in
;-odac        -iadc      ;;RT audio I/O
; For Non-realtime output leave only the line below:
-o modmatrix.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 441
```

```

ksmps = 100
nchnls = 2
Odbfs = 1

; basic waveforms
giSine ftgen 0, 0, 65537, 10, 1 ; sine wave
giSaw ftgen 0, 0, 4097, 7, 1, 4096, -1 ; saw (linear)
giSoftSaw ftgen 0, 0, 65537, 30, giSaw, 1, 10 ; soft saw (only 10 first harmonics)

; modmatrix tables
giMaxNumParam = 128
giMaxNumMod = 32
giParam_In ftgen 0, 0, giMaxNumParam, 2, 0 ; input parameters table
; output parameters table (parameter values with added modulators)
giParam_Out ftgen 0, 0, giMaxNumParam, 2, 0
giModulators ftgen 0, 0, giMaxNumMod, 2, 0 ; modulators table
; modulation scaling and routing (mod matrix) table, start with empty table
giModScale ftgen 0, 0, giMaxNumParam*giMaxNumMod, -2, 0

;*****
; generate the modulator signals
;*****
instr 1

; LFO1, 1.5 Hz, normalized range (0.0 to 1.0)
kLFO1 oscil 0.5, 1.5, giSine ; generate LFO signal
kLFO1 = kLFO1+0.5 ; offset

; LFO2, 0.4 Hz, normalized range (0.0 to 1.0)
kLFO2 oscil 0.5, 0.4, giSine ; generate LFO signal
kLFO2 = kLFO2+0.5 ; offset

; write modulators to table
tablew kLFO1, 0, giModulators
tablew kLFO2, 1, giModulators

endin

;*****
; set parameter values
;*****
instr 2

; Here we can set the parameter values
icps1 = p4
icps2 = p5
icutoff = p6

; write parameters to table
tableiw icps1, 0, giParam_In
tableiw icps2, 1, giParam_In
tableiw icutoff, 2, giParam_In

endin

;*****
; mod matrix edit
;*****
instr 3

; Here we can write to the modmatrix table by using tablew or tableiw

iLfo1ToCps1 = p4
iLfo1ToCps2 = p5
iLfo1ToCutoff = p6
iLfo2ToCps1 = p7

```

```

iLfo2ToCps2 = p8
iLfo2ToCutoff = p9

***(must update to enable changes)
ktrig init 1
chnset ktrig, "modulatorUpdateFlag"
ktrig = 0

endin

;*****
; mod matrix
;*****
instr 4

; get the update flag
kupdate chnget "modulatorUpdateFlag"

; run the mod matrix
inum_mod = 2
inum_parm = 3
modmatrix giParam_Out, giModulators, giParam_In, \
giModScale, inum_mod, inum_parm, kupdate

; and reset the update flag
chnset 0, "modulatorUpdateFlag" ; reset the update flag

endin

;*****
; audio generator to test values
;*****
instr 5

; basic parameters
iamp = ampdbfs(-5)

; read modulated parameters from table
kcps1 table 0, giParam_Out
kcps2 table 1, giParam_Out
kcutoff table 2, giParam_Out

; set filter parameters
kCF_freq1 = kcps1*kcutoff
kCF_freq2 = kcps2*kcutoff
kReso = 0.7
kDist = 0.3

; oscillators and filters
a1 oscili iamp, kcps1, giSoftSaw
a1 lpf18 a1, kCF_freq1, kReso, kDist

a2 oscili iamp, kcps2, giSoftSaw
a2 lpf18 a2, kCF_freq2, kReso, kDist

outs a1, a2

endin

```

```
</CsInstruments>
<CsScore>

;*****
; set initial parameters
; cps1 cps2 cutoff
i2 0 1 400 800 3

;*****
; set modmatrix values
; lfo1ToCps1 lfo1ToCps2 lfo1ToCut lfo2ToCps1 lfo2ToCps2 lfo2ToCut
i3 0 1 40 0 -2 -50 100 3

;*****
; start "always on" instruments
#define SCORELEN # 20 # ; set length of score

i1 0 $SCORELEN ; start modulators
i4 0 $SCORELEN ; start mod matrix
i5 0 $SCORELEN ; start audio oscillator

e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*Linear Algebra Opcodes, Vectorial Opcodes, tablew.*

## Credits

By:

Oeyvind Brandtsegg and Thom Johansen

New in version 5.12

# monitor

monitor — Returns the audio spout frame.

## Description

Returns the audio spout frame (if active), otherwise it returns zero.

## Syntax

```
aout1 [,aout2 ... aoutX] monitor  
aarra monitor
```

## Performance

In the array form it reads all channels to a one dimensional array.

This opcode can be used for monitoring the output signal from csound. It should not be used for processing the signal further.

See the entry for the *fout* opcode for an example of usage of *monitor*.

## Examples

Here is an example of the monitor opcode. It uses the file *monitor.csd* [examples/monitor.csd].

### Example 571. Example of the monitor opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
-odac      ;;realtime audio out  
;-iadc     ;;uncomment -iadc if realtime audio input is needed too  
; For Non-realtime ouput leave only the line below:  
; -o monitor.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>  
  
sr = 44100  
ksmps = 32  
nchnls = 2  
0dbfs = 1  
  
giSine ftgen 0, 0, 2^10, 10, 1  
  
instr 1  
  
asig poscil3 .5, 880, giSine  
;write a raw file: 32 bits with header  
fout "fout_880.wav", 15, asig  
outs asig, asig
```

```
    endin

    instr 2

    klfo lfo 1, 2, 0
    asig poscil3 .5*klfo, 220, giSine
    ;write an aiff file: 32 bits with header
        fout "fout_aif.aiff", 25, asig
    ;        fout "fout_all3.wav", 14, asig
        outs asig, asig

    endin

    instr 99 ;read the stereo csound output buffer

    allL, allR monitor
    ;write the output of csound to an audio file
    ;to a wav file: 16 bits with header
        fout "fout_all.wav", 14, allL, allR

    endin
</CsInstruments>
<CsScore>

i 1 0 2
i 2 0 3
i 99 0 3
e
</CsScore>
</CsoundSynthesizer>
```

## See also

*fout*, the *Mixer opcodes* and the *Zak Patching System*.

## Credits

Istvan Varga 2006

John ffitch 2016 for array form, new in 6.08

# moog

moog — An emulation of a mini-Moog synthesizer.

## Description

An emulation of a mini-Moog synthesizer.

## Syntax

```
ares moog kamp, kfreq, kfiltq, kfiltrate, kvibf, kvamp, iafn, iwfn, ivfn
```

## Initialization

*iafn*, *iwfn*, *ivfn* -- three table numbers containing the attack waveform (unlooped), the main looping waveform, and the vibrato waveform. The files *mandpluk.aiff* [examples/mandpluk.aiff] and *impuls20.aiff* [examples/impuls20.aiff] are suitable for the first two, and a sine wave for the last.



### Note

The files “mandpluk.aiff” and “impuls20.aiff” are also available at <ftp://ftp.c-s.bath.ac.uk/pub/dream/documentation/sounds/modelling/>.

## Performance

*kamp* -- Amplitude of note.

*kfreq* -- Frequency of note played.

*kfiltq* -- Q of the filter, in the range 0.8 to 0.9

*kfiltrate* -- rate control for the filter in the range 0 to 0.0002

*kvibf* -- frequency of vibrato in Hertz. Suggested range is 0 to 12

*kvamp* -- amplitude of the vibrato

## Examples

Here is an example of the moog opcode. It uses the file *moog.csd* [examples/moog.csd], *mandpluk.aiff* [examples/mandpluk.aiff], and *impuls20.aiff* [examples/impuls20.aiff].

### Example 572. Example of the moog opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;;realtime audio out
```

```
;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
;-o moog.wav -W   ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kfreq = cpspch(p4)
kfiltq = p5
kfiltrate = 0.0002
kvibf = 5
kvamp = .01
;low volume is needed
asig moog .15, kfreq, kfiltq, kfiltrate, kvibf, kvamp, 1, 2, 3
outs asig, asig

endin
</CsInstruments>
<CsScore>

f 1 0 8192 1 "mandpluk.aiff" 0 0 0
f 2 0 256 1 "impuls20.aiff" 0 0 0
f 3 0 256 10 1 ; sine

i 1 0 3 6.00 .1
i 1 + 3 6.05 .89
i 1 + 3 6.09 .50
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitich (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

# moogladder

moogladder — Moog ladder lowpass filter.

## Description

Moogladder is a new digital implementation of the Moog ladder filter based on the work of Antti Huovilainen, described in the paper "Non-Linear Digital Implementation of the Moog Ladder Filter" (Proceedings of DaFX04, Univ of Napoli). This implementation is probably a more accurate digital representation of the original analogue filter.

## Syntax

```
asig moogladder ain, kcf, kres[, istor]
asig moogladder ain, acf, kres[, istor]
asig moogladder ain, kcf, ares[, istor]
asig moogladder ain, acf, ares[, istor]
```

## Initialization

*istor* --initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*asig* -- input signal.

*kcf/acf* -- filter cutoff frequency

*kres/ares* -- resonance, generally  $< 1$ , but not limited to it. Higher than 1 resonance values might cause aliasing, analogue synths generally allow resonances to be above 1.

## Examples

Here is an example of the moogladder opcode. It uses the file *moogladder.csd* [examples/moogladder.csd].

### Example 573. Example of the moogladder opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o moogladder.wav -W ;; for file output any platform
</CsOptions>
```

```
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kfe expseg 500, p3*0.9, 1800, p3*0.1, 3000
asig buzz 1, 100, 20, 1
kres line .1, p3, .99 ;increase resonance
afil moogladder asig, kfe, kres
      outs afil, afil

endin
</CsInstruments>
<CsScore>
f 1 0 4096 10 1

i 1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
January 2005

New plugin in version 5

January 2005.

Audio rate parameters introduced in version 6.02

October 2013.

# moogladder2

moogladder2 — Moog ladder lowpass filter.

## Description

Moogladder2 is a new digital implementation of the Moog ladder filter based on the work of Antti Huovilainen, described in the paper "Non-Linear Digital Implementation of the Moog Ladder Filter" (Proceedings of DaFX04, Univ of Napoli). This implementation uses approximations to the tanh function and so is faster but less accurate than *moogladder*.

## Syntax

```
asig moogladder2 ain, kcf, kres[, istor]
asig moogladder2 ain, acf, kres[, istor]
asig moogladder2 ain, kcf, ares[, istor]
asig moogladder2 ain, acf, ares[, istor]
```

## Initialization

*istor* --initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*asig* -- input signal.

*kcf/acf* -- filter cutoff frequency

*kres/ares* -- resonance, generally  $< 1$ , but not limited to it. Higher than 1 resonance values might cause aliasing, analogue synths generally allow resonances to be above 1.

## Examples

Here is an example of the moogladder2 opcode. It uses the file *moogladder2.csd* [examples/moogladder2.csd].

### Example 574. Example of the moogladder2 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o moogladder2.wav -W ;; for file output any platform
```

```
</CsOptions>
<CsInstruments>

  sr = 44100
  ksmps = 32
  nchnls = 2
  0dbfs = 1

  instr 1

  kfe expseg 500, p3*0.9, 1800, p3*0.1, 3000
  asig buzz 1, 100, 20, 1
  kres line .1, p3, .99 ;increase resonance
  afil moogladder2 asig, kfe, kres
      outs afil, afil

  endin
</CsInstruments>
<CsScore>
f 1 0 4096 10 1

i 1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini

Author: John ffitc

April 2016

New in version 6.07

# moogvcf

`moogvcf` — A digital emulation of the Moog diode ladder filter configuration.

## Description

A digital emulation of the Moog diode ladder filter configuration.

## Syntax

```
ares moogvcf asig, xfco, xres [,iscale, iskip]
```

## Initialization

*iscale* (optional, default=1) -- internal scaling factor. Use if *asig* is not in the range +/-1. Input is first divided by *iscale*, then output is multiplied *iscale*. Default value is 1. (New in Csound version 3.50)

*iskip* (optional, default=0) -- if non zero skip the initialisation of the filter. (New in Csound version 4.23f13 and 5.0)

## Performance

*asig* -- input signal

*xfco* -- filter cut-off frequency in Hz. As of version 3.50, may i-,k-, or a-rate.

*xres* -- amount of resonance. Self-oscillation occurs when *xres* is approximately one. As of version 3.50, may a-rate, i-rate, or k-rate.

*moogvcf* is a digital emulation of the Moog diode ladder filter configuration. This emulation is based loosely on the paper “Analyzing the Moog VCF with Considerations for Digital Implementation” by Stilson and Smith (CCRMA). This version was originally coded in Csound by Josep Comajuncosas. Some modifications and conversion to C were done by Hans Mikelson.



### Warning

Before version 6.02 this filter required that the input signal be normalized to one. This can be easily achieved using *Odbfs*, like this:

```
ares moogvcf asig, kfco, kres, Odbfs
```

You can also use *moogvcf2* which defaults scaling to *Odbfs*.

## Examples

Here is an example of the `moogvcf` opcode. It uses the file *moogvcf.csd* [examples/moogvcf.csd].

### Example 575. Example of the `moogvcf` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o moogvcf.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
; iscale does not need to be set here because already 0dbfs = 1
aout vco .3, 220, 1 ; Use a nice sawtooth waveform.
kfco line 200, p3, 2000 ; filter-cutoff frequency from .2 to 2 KHz
krez init p4
asig moogvcf aout, kfco, krez
    outs asig, asig

endin
</CsInstruments>
<CsScore>
;a sine wave
f 1 0 16384 10 1

i 1 0 3 .1
i 1 + 3 .7
i 1 + 3 .95
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*moogvcf2, biquad, rezzv*

## Credits

Author: Hans Mikelson  
October 1998

New in Csound version 3.49

# moogvcf2

moogvcf2 — A digital emulation of the Moog diode ladder filter configuration.

## Description

A digital emulation of the Moog diode ladder filter configuration.

## Syntax

```
ares moogvcf2 asig, xfco, xres [,iscale, iskip]
```

## Initialization

*iscale* (optional, default=0dBfs) -- internal scaling factor, as the operation of the code requires the signal to be in the range +/-1. Input is first divided by *iscale*, then output is multiplied by *iscale*.

*iskip* (optional, default=0) -- if non zero skip the initialisation of the filter.

## Performance

*asig* -- input signal

*xfco* -- filter cut-off frequency in Hz. which may be i-,k-, or a-rate.

*xres* -- amount of resonance. Self-oscillation occurs when *xres* is approximately one. May be a-rate, i-rate, or k-rate.

*moogvcf2* is a digital emulation of the Moog diode ladder filter configuration. This emulation is based loosely on the paper “Analyzing the Moog VCF with Considerations for Digital Implementation” by Stilson and Smith (CCRMA). This version was originally coded in Csound by Josep Comajuncosas. Some modifications and conversion to C were done by Hans Mikelson and then adjusted.

*moogvcf2* is identical to *moogvcf*, except that the *iscale* parameter defaults to *0dbfs* instead of 0, guaranteeing that amplitude will usually be OK.

## Examples

Here is an example of the moogvcf2 opcode. It uses the file *moogvcf2.csd* [examples/moogvcf2.csd].

### Example 576. Example of the moogvcf2 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o moogvcf2.wav -W ;; for file output any platform
```

```
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

aout diskin2 "beats.wav", 1, 0, 1
kfco line 100, p3, 10000 ;filter-cutoff
krez init p4
asig moogvcf2 aout, kfco, krez
outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 4 .1
i 1 + 4 .6
i 1 + 4 .9
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*moogvcf, biquad, rezy*

## Credits

Author: Hans Mikelson and John ffitch  
October 1998/ July 2006

New in Csound version 5.03

# moscil

moscil — Sends a stream of the MIDI notes.

## Description

Sends a stream of the MIDI notes.

## Syntax

```
moscil kchn, knum, kvel, kdur, kpause
```

## Performance

*kchn* -- MIDI channel number (1-16)

*knum* -- note number (0-127)

*kvel* -- velocity (0-127)

*kdur* -- note duration in seconds

*kpause* -- pause duration after each noteoff and before new note in seconds

*moscil* and *midion* are the most powerful MIDI OUT opcodes. *moscil* (MIDI oscil) plays a stream of notes of *kdur* duration. Channel, pitch, velocity, duration and pause can be controlled at k-rate, allowing very complex algorithmically generated melodic lines. When current instrument is deactivated, the note played by current instance of *moscil* is forcedly truncated.

Any number of *moscil* opcodes can appear in the same Csound instrument, allowing a counterpoint-style polyphony within a single instrument.

## Examples

Here is an example of the *moscil* opcode. It uses the file *moscil.csd* [examples/moscil.csd].

### Example 577. Example of the *moscil* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

This example generates a stream of notes for every note received on the MIDI input. It generates MIDI notes on csound's MIDI output, so be sure to connect something.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      -M0  -Q1;;;RT audio I/O with MIDI in
</CsOptions>
<CsInstruments>

sr = 44100
```

```
kr = 4410
ksmps = 10
nchnls = 2

; Example by Giorgio Zucco 2007

instr 1 ;Triggered by MIDI notes on channel 1

  inote notnum
  ivel  veloc

  kpitch = 40
  kfreq = 2

  kdur = .04
  kpause = .1

  k1 lfo kpitch, kfreq,5

  ;plays a stream of notes of kdur duration on MIDI channel 1
  moscil 1, inote + k1, ivel, kdur, kpause

endin

</CsInstruments>
<CsScore>
; Dummy ftable
f0 60
</CsScore>
</CsoundSynthesizer>
```

## See Also

*midion, midion2, noteon, noteoff, noteondur, noteondur2*

## Credits

Author: Gabriel Maldonado  
Italy  
May 1997

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# mp3in

mp3in — Reads mono or stereo audio data from an external MP3 file.

## Description

Reads mono or stereo audio data from an external MP3 file.

## Syntax

```
ar1, ar2 mp3in ifilcod[, iskptim, iformat, iskipinit, ibufsize]
ar1 mp3in ifilcod[, iskptim, iformat, iskipinit, ibufsize]
```

## Initialization

*ifilcod* -- integer or character-string denoting the source soundfile name. An integer denotes the file soundin.filcod ; a character-string (in double quotes, spaces permitted) gives the filename itself, optionally a full pathname. If not a full path, the named file is sought first in the current directory, then in that given by the environment variable *SSDIR* (if defined) then by *SFDIR*.

*iskptim* (optional) -- time in seconds of input sound to be skipped. The default value is 0.

*iformat* (optional) -- specifies the audio data file format: currently not implemented and always defaults to stereo.

*iskipinit* (optional) -- switches off all initialisation if non zero (default =0).

*ibuffersize* (optional) -- sets the internal buffer size for reading. If the value is omitted, zero or negative it defaults to 4096 bytes.

## Performance

Reads audio data from an external MP3 file.

## Examples

Here is an example of the mp3in opcode. It uses the file *mp3in.csd* [examples/mp3in.csd].

### Example 578. Example of the mp3in opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o mp3in.wav -W ;; for file output any platform
</CsOptions>
```

```
<CsInstruments>

  sr = 44100
  ksmps = 32
  nchnls = 2
  0dbfs = 1

  instr 1

  iskptim = .3
  ibufsize = 64
  ar1, ar2 mp3in "beats.mp3", iskptim, 0, 0, ibufsize
    outs ar1, ar2

  endin
</CsInstruments>
<CsScore>

  i 1 0 2
  e
</CsScore>
</CsSoundSynthesizer>
```

## See Also

*diskin, ins, in, inh, inh, ino, inq, mp3len, soundin*

## Credits

Author: John fitch  
Codemist Ltd  
2009

New in version 5.11

Mono added in 6.05

# mp3len

mp3len — Returns the length of an MP3 sound file.

## Description

Returns the length of an MP3 sound file.

## Syntax

```
ir mp3len ifilcod
```

## Initialization

*ifilcod* -- sound file to be queried

## Performance

*mp3len* returns the length of the sound file *ifilcod* in seconds.

## Examples

Here is an example of the mp3len opcode. It uses the file *mp3len.csd* [examples/mp3len.csd].

### Example 579. Example of the mp3len opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
;-o mp3len.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ilen mp3len p4      ;calculate length of mp3 file
print ilen

asigL, asigR mp3in p4
outs asigL, asigR

endin
</CsInstruments>
<CsScore>
```

```
i 1 0 30 "XORNOT_jul-14-05.mp3" ; long signal  
e  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*mp3in*

## Credits

Author: John fitch

Feb 2011

New in Csound version 5.14

# mp3scal

mp3scal — Phase-locked vocoder processing with onset detection/processing, 'tempo-scaling'.

## Description

*mp3scal* implements phase-locked vocoder processing from mp3-format disk files, resampling if necessary.

This opcode allows for time and frequency-independent scaling. Time is advanced internally. The quality of the effect is generally improved with phase locking switched on.

*mp3scal* will also scale pitch, independently of frequency, using a transposition factor (k-rate).

## Syntax

```
asig, asig2, ktime mp3scal Sfile,ktimescal,kpitch, kamp [,iskip,ifftsize, idecim, ilock]
```

## Initialization

*Sfile* -- source soundfile stereo mp3 files.

*ifftsize* -- FFT size (power-of-two), defaults to 2048.

*idecim* -- decimation, defaults to 4 (meaning hopsize = fftsize/4)

*iskip* -- skiptime in seconds, defaults to 1.

*ilock* -- 0 or 1, to switch phase-locking on/off, defaults to 1.

## Performance

*ktimescal* -- timescaling ratio, < 1 stretch, > 1 contract. Non-negative numbers only.

*kamp* -- amplitude scaling

*kpitch* -- grain pitch scaling (1=normal pitch, < 1 lower, > 1 higher)

*ktime* -- time stamp

*asig*, *asig2* -- stereo output signals

## Examples

Here is an example of the mp3scal opcode. It uses the file *mp3scal.csd* [examples/mp3scal.csd].

### Example 580. Example of the mp3scal opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
nchnls=2
ksmps=64
sr=44100
instr 1
SFile = p4
p3 = mp3len(SFile)/p5
a1,a2,k2 mp3scal SFile,p5,1,1
outs a1,a2

endin

</CsInstruments>
<CsScore>
i1.1 0 1 "beats.mp3" .75
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
April 2016

New in version 6.07

April 2016.

# mpulse

mpulse — Generates a set of impulses.

## Description

Generates a set of impulses of amplitude *kamp* separated by *kintvl* seconds (or samples if *kintvl* is negative). The first impulse is generated after a delay of *ioffset* seconds.

## Syntax

```
ares mpulse kamp, kintvl [, ioffset]
```

## Initialization

*ioffset* (optional, default=0) -- the delay before the first impulse. If it is negative, the value is taken as the number of samples, otherwise it is in seconds. Default is zero.

## Performance

*kamp* -- amplitude of the impulses generated

*kintvl* -- Interval of time in seconds (or samples if *kintvl* is negative) to the next pulse.

After the initial delay, an impulse of *kamp* amplitude is generated as a single sample. Immediately after generating the impulse, the time of the next one is determined from the value of *kintvl* at that precise moment. This means that any changes in *kintvl* between impulses are discarded. If *kintvl* is zero, there is an infinite wait to the next impulse. If *kintvl* is negative, the interval is counted in number of samples rather than seconds.

## Examples

Here is an example of the mpulse opcode. It uses the file *mpulse.csd* [examples/mpulse.csd].

### Example 581. Example of the mpulse opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o mpulse.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
```

```
ksmps = 10
nchnls = 1

gkfreq init 0.1

instr 1
  kamp = 10000

  a1 mpulse kamp, gkfreq
  out a1
endin

instr 2
; Assign the value of p4 to gkfreq
gkfreq init p4
endin
</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 11
i 2 2 1 0.05
i 2 4 1 0.01
i 2 6 1 0.005
; only last notes are audible
i 2 8 1 0.003
i 2 10 1 0.002

e

</CsScore>
</CsoundSynthesizer>
```

## See Also

Another example of how to use mpulse can be found here: *mode*

## Credits

Written by John fitch.

New in version 4.08

# mrtmsg

mrtmsg — Send system real-time messages to the MIDI OUT port.

## Description

Send system real-time messages to the MIDI OUT port.

## Syntax

```
mrtmsg imgstype
```

## Initialization

*imgstype* -- type of real-time message:

- 1 sends a START message (0xFA);
- 2 sends a CONTINUE message (0xFB);
- 0 sends a STOP message (0xFC);
- -1 sends a SYSTEM RESET message (0xFF);
- -2 sends an ACTIVE SENSING message (0xFE)

## Performance

Sends a real-time message once, in init stage of current instrument. *imgstype* parameter is a flag to indicate the message type.

## See Also

*mclock*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

# mtof

mtof — Convert a midi to frequency

## Description

Convert a midi note number value to cycles per second, taking global value of A4 into account.

## Syntax

```
ifreq mtof imidi  
kfreq mtof kmidi
```

## Performance

*kmidi* / *imidi* -- Midi note number

*kfreq* / *ifreq* -- Frequency corresponding to midi note value

## Examples

Here is an example of the mtof opcode. It uses the file *mtof-ftom.csd* [examples/mtof-ftom.csd].

### Example 582. Example of the mtof opcode.

```
<CsoundSynthesizer>  
<CsOptions>  
</CsOptions>  
<CsInstruments>  
  
sr = 44100  
ksmps = 128  
nchnls = 2  
0dbfs = 1.0  
A4  
  
instr 1  
  kfreq = mtof(69)  
  printks2 "midi 69 -> %f\n", kfreq  
  
  kmidi = ftom(442)  
  printks2 "freq 442 -> %f\n", kmidi  
  
  kfreq = mtof(kmidi)  
  printks "midi %f -> %f\n", 1, kmidi, kfreq  
  
  imidi = ftom:i(440)  
  print imidi  
  
  ifreq = mtof:i(60)  
  print ifreq  
  
turnoff
```

```
endin  
  
</CsInstruments>  
<CsScore>  
i 1 0 10  
  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*cpsmidim, ftom, mton, ntom*

## Credits

By: Eduardo Moguillansky 2017

New in version 6.09

# mton

mton — Convert midi note number to string note name

## Description

Convert midi note number to string note name, with an accuracy of 1 cent.

## Syntax

```
Snote mton kmidi
```

```
Snote mton imidi
```

## Performance

*kmidi / imidi* -- Midi note number

*Snote* -- Note name

Example of note names:

midi	note name
60	4C
60.4	4C+40
60.5	4C+
60.9	4Db-10
61	4C#
61.5	4D-

## Examples

Here is an example of the mton opcode. It uses the file *mton-ntom.csd* [examples/mton-ntom.csd].

### Example 583. Example of the ftom opcode.

```
<CsoundSynthesizer>
<CsOptions>
--nosound
</CsOptions>
<CsInstruments>

instr 1
  S4 mton ntom("7D+63")
  puts S4, 1

  S1 mton 60
  puts S1, 1

  S2 mton ftom(442)
  puts S2, 1

  S3 = mton(48.25)
```

```
puts S3, 1

k1 = ntom("4C")
printk2 k1

i2 ntom "4E"
print i2

S5 = mton(ntom("4G+"))
puts S5, 1

turnoff
endin

instr 2
k1 = ntom("4Eb-31")
printk2 k1
i0 ntom "4C+"
print i0
i1 = ntom:i("4A")
print i1
turnoff
endin

</CsInstruments>
<CsScore>

i 1 0 1
i 2 0 1

</CsScore>
</CsoundSynthesizer>
```

## See Also

*cpsmidinn, mtof, ftoM, ntom*

## Credits

By: Eduardo Mogueillansky 2017

New in version 6.09

# multitap

multitap — Multitap delay line implementation.

## Description

Multitap delay line implementation.

## Syntax

```
ares multitap asig [, itime1, igain1] [, itime2, igain2] [...]
```

## Initialization

The arguments *itime* and *igain* set the position and gain of each tap.

The delay line is fed by *asig*.

## Examples

Here is an example of the multitap opcode. It uses the file *multitap.csd* [examples/multitap.csd]

### Example 584. Example of the multitap opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o multitap.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

ga1 init 0

instr 1

asig diskin2 "beats.wav", 1,0
outs asig, asig

ga1 = ga1+asig
endin

instr 2

asig multitap ga1, 1.2, .5, 1.4, .2
outs asig, asig

ga1 = 0
endin
```

```
</CsInstruments>
<CsScore>

i 1 .5 .2 ; short sound
i 2 0 3 ; echoes
e
</CsScore>
</CsoundSynthesizer>
```

This results in two delays, one with length of 1.2 and gain of .5, and one with length of 1.4 and gain of .2.

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1996

# mute

mute — Mutes/unmutes new instances of a given instrument.

## Description

Mutes/unmutes new instances of a given instrument.

## Syntax

```
mute insnum [, iswitch]
```

```
mute "insname" [, iswitch]
```

## Initialization

*insnum* -- instrument number. Equivalent to *p1* in a score *i statement*.

*"insname"* -- A string (in double-quotes) representing a named instrument.

*iswitch* (optional, default=0) -- represents a switch to mute/unmute an instrument. A value of 0 will mute new instances of an instrument, other values will unmute them. The default value is 0.

## Performance

All new instances of instrument *inst* will be muted (*iswitch* = 0) or unmuted (*iswitch* not equal to 0). There is no difficulty with muting muted instruments or unmuting unmuted instruments. The mechanism is the same as used by the score *q statement*. For example, it is possible to mute in the score and unmute in some instrument.

Muting/Unmuting is indicated by a message (depending on message level).

## Examples

Here is an example of the mute opcode. It uses the file *mute.csd* [examples/mute.csd].

### Example 585. Example of the mute opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o mute.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
```

```
nchnls = 2
odbfs = 1

; Mute Instrument #2.
mute 2
; Mute Instrument three.
mute "three"

instr 1

a1 oscils 0.2, 440, 0
  outs a1, a1
endin

instr 2 ; gets muted

a1 oscils 0.2, 880, 0
  outs a1, a1
endin

instr three ; gets muted

a1 oscils 0.2, 1000, 0
  outs a1, a1
endin

</CsInstruments>
<CsScore>

i 1 0 1
i 2 0 1
i "three" 0 1
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

New in version 4.22

# mvchpf

mvchpf — Moog voltage-controlled highpass filter emulation.

## Description

Mvchpf is an digital implementation of the 4th-order (24 dB/oct) Moog high-pass filter, originally written by Fons Andriaensen. According to the author, mvchpf "...is based on the voltage controlled highpass filter by Robert Moog, again with some attention to the nonlinear effects."

## Syntax

```
asig mvchpf ain, xcf[, istor]
```

## Initialization

*istor* --initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*ain* -- input signal.

*xcf* -- filter cutoff frequency. The useful range is around six octaves below and above middle C (pch 8.00).

## Examples

Here is an example of the mvchpf opcode. It uses the file *mvchpf.csd* [examples/mvchpf.csd].

### Example 586. Example of the mvchpf opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
0dbfs = 1

instr 1
  kenv linen p4,0.1,p3,0.1
  ain rand kenv
  kfr expon 220, p3, 1760
  asig mvchpf ain,kfr
  out asig
endin

</CsInstruments>
<CsScore>
i1 0 5 0.9
```

```
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Author: Fons Andriaensen,  
ported to Csound by Victor Lazzarini  
January 2016

New in version 6.07

January 2016.

# mvclpf1

mvclpf1 — Moog voltage-controlled lowpass filter emulation.

## Description

Mvclpf1 is an digital implementation of the 4th-order (24 dB/oct) Moog ladder filter originally written by Fons Andriaensen. According to the author, mvclpf1 "is a fairly simple design, and it does not even pretend to come close the 'real thing'. It uses a very crude approximation of the non-linear resistor in the first filter section only. [...] [I]t's [a] cheap (in terms of CPU usage) general purpose 24 dB/oct lowpass filter that could be useful".

## Syntax

```
asig mvclpf1 ain, xcf, xres[,istor]
```

## Initialization

*istor* --initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*ain* -- input signal.

*xcf* -- filter cutoff frequency. The useful range is around six octaves below and above middle C (pch 8.00).

*xres* -- resonance, limited to the interval [0,1].

## Examples

Here is an example of the mvclpf1 opcode. It uses the file *mvclpf1.csd* [examples/mvclpf1.csd].

### Example 587. Example of the mvclpf1 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
0dbfs = 1

instr 1
  kenv linen p4,0.1,p3,0.1
  ain rand kenv
  kfr expon 220, p3, 1760
  asig mvclpf1 ain,kfr,0.9
    out asig
endin
```

```
</CsInstruments>  
<CsScore>  
i1 0 5 0.9  
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Author: Fons Andriaensen,  
ported to Csound by Victor Lazzarini  
January 2016

New in version 6.07

January 2016.

# mvclpf2

mvclpf2 — Moog voltage-controlled lowpass filter emulation.

## Description

Mvclpf2 is an digital implementation of the 4th-order (24 dB/oct) Moog ladder filter originally written by Fons Andriaensen. According to the author, mvclpf2 "uses five non-linear elements, in the input and in all four filter sections. It works by using the derivative of the nonlinearity (for which  $1 / (1 + x * x)$  is reasonable approximation). The main advantage of this is that only one evaluation of the non-linear function is required for each section".

## Syntax

```
asig mvclpf2 ain, xcf, xres[, istor]
```

## Initialization

*istor* --initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*ain* -- input signal.

*xcf* -- filter cutoff frequency. The useful range is around six octaves below and above middle C (pch 8.00).

*xres* -- resonance, limited to the interval [0,1].

## Examples

Here is an example of the mvclpf2 opcode. It uses the file *mvclpf2.csd* [examples/mvclpf2.csd].

### Example 588. Example of the mvclpf2 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
0dbfs = 1

instr 1
  kenv linen p4,0.1,p3,0.1
  ain rand kenv
  kfr expon 220, p3, 1760
  asig mvclpf1 ain,kfr,0.9
    out asig
endin
```

```
</CsInstruments>  
<CsScore>  
i1 0 5 0.9  
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Author: Fons Andriaensen,  
ported to Csound by Victor Lazzarini  
January 2016

New in version 6.07

January 2016.

# mvclpf3

mvclpf3 — Moog voltage-controlled lowpass filter emulation.

## Description

Mvclpf3 is an digital implementation of the 4th-order (24 dB/oct) Moog ladder filter originally written by Fons Andriaensen. According to the author, mvclpf3 "is based on mvclpf2 , with two differences. It uses the the technique described by Stilson and Smith to extend the constant-Q range, and the internal sample frequency is doubled, giving a better approximation to the non-linear behaviour at high frequencies. This version has high Q over the entire frequency range and will oscillate up to above 10 kHz, while the two others show a decreasing Q at high frequencies. Mvclpf3 is reasonably well tuned, and can be 'played' as a VCO up to at least 5 kHz".

## Syntax

```
asig mvclpf3 ain, xcf, xres[, istor]
```

## Initialization

*istor* --initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*ain* -- input signal.

*xcf*-- filter cutoff frequency. The useful range is around six octaves below and above middle C (pch 8.00).

*xres* -- resonance, limited to the interval [0,1].

## Examples

Here is an example of the mvclpf3 opcode. It uses the file *mvclpf3.csd* [examples/mvclpf3.csd].

### Example 589. Example of the mvclpf3 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
0dbfs = 1

instr 1
  kenv linen p4,0.1,p3,0.1
  ain rand kenv
  kfr expon 220, p3, 1760
```

```
asig mvclpf3 ain,kfr,0.9
  out asig
endin

</CsInstruments>
<CsScore>
i1 0 5 0.9
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Fons Andriaensen,  
ported to Csound by Victor Lazzarini  
January 2016

New in version 6.07

January 2016.

# mvclpf4

mvclpf4 — Moog voltage-controlled lowpass filter emulation.

## Description

Mvclpf4 is a digital implementation of the 4th-order (24 dB/oct) Moog ladder filter originally written by Fons Andriaensen. It is a version of the mvclpf3 opcode with four outputs, for 6dB, 12dB, 18dB, and 24 dB/octave responses.

## Syntax

```
asig1,asig2,asig3,asig4 mvclpf4 ain, xcf, xres[, istor]
```

## Initialization

*istor* --initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*asig1* -- 6dB/oct low-pass response output.

*asig2* -- 12dB/oct low-pass response output.

*asig3* -- 18dB/oct low-pass response output..

*asig4* -- 24dB/oct low-pass response output.

*ain* -- input signal.

*xcf* -- filter cutoff frequency. The useful range is around six octaves below and above middle C (pch 8.00).

*xres* -- resonance, limited to the interval [0,1].

## Examples

Here is an example of the mvclpf4 opcode. It uses the file *mvclpf4.csd* [examples/mvclpf4.csd].

### Example 590. Example of the mvclpf4 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
-odac  
</CsOptions>  
<CsInstruments>  
0dbfs = 1
```

```
instr 1
  asig[] init 4
  kenv linen p4,0.1,p3,0.1
  ain rand kenv
  kfr expon 220, p3, 1760
  asig[0],asig[1],asig[2],asig[3] mvclpf4 ain,kfr,0.9
  out asig[p5]
endin

</CsInstruments>
<CsScore>
i1 0 5 0.9 0
i1 + 5 0.9 1
i1 + 5 0.9 2
i1 + 5 0.9 3
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Fons Andriaensen,  
ported to Csound by Victor Lazzarini  
January 2016

New in version 6.07

January 2016.

## mxadsr

`mxadsr` — Calculates the classical ADSR envelope using the *expsegr* mechanism.

## Description

Calculates the classical ADSR envelope using the *expsegr* mechanism.

## Syntax

```
ares mxadsr iatt, idec, islev, irel [, idel] [, ireltim]
```

```
kres mxadsr iatt, idec, islev, irel [, idel] [, ireltim]
```

## Initialization

*iatt* -- duration of attack phase

*idec* -- duration of decay

*islev* -- level for sustain phase

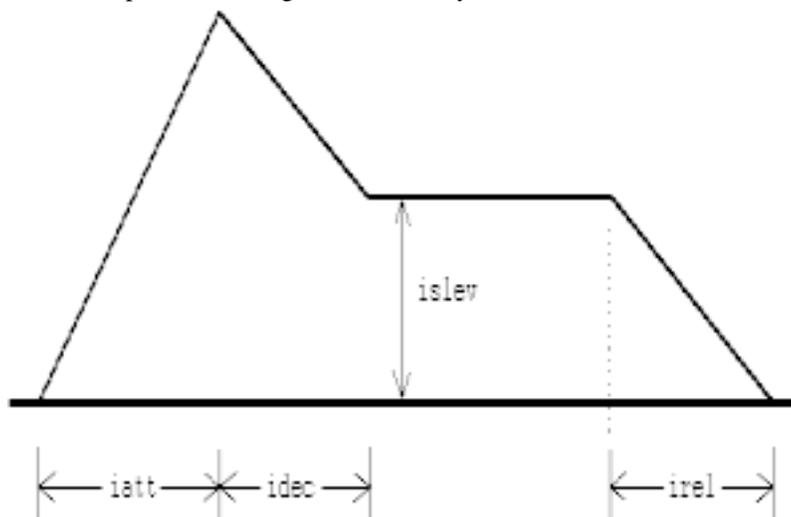
*irel* -- duration of release phase

*idel* (optional, default=0) -- period of zero before the envelope starts

*ireltim* (optional, default=-1) -- Control release time after receiving a MIDI noteoff event. If less than zero, the longest release time given in the current instrument is used. If zero or more, the given value will be used for release time. Its default value is -1. (New in Csound 3.59 - not yet properly tested)

## Performance

The envelope is in the range 0 to 1 and may need to be scaled further. The envelope may be described as:



Picture of an ADSR envelope.

The length of the sustain is calculated from the length of the note. This means *adsr* is not suitable for use with MIDI events. The opcode *madsr* uses the *linsegr* mechanism, and so can be used in MIDI applications. The opcode *mxadsr* is identical to *madsr* except it uses exponential, rather than linear, line segments.

You can use other pre-made envelopes which start a release segment upon receiving a note off message, like *linsegr* and *expsegr*, or you can construct more complex envelopes using *xtratim* and *release*. Note that you don't need to use *xtratim* if you are using *mxadsr*, since the time is extended automatically.

*mxadsr* is new in Csound version 3.51.

## Examples

Here is an example of the *mxadsr* opcode. It uses the file *mxadsr.csd* [examples/mxadsr.csd].

### Example 591. Example of the *mxadsr* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0   ;;realtime audio out and realtime midi in
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o mxadsr.wav -W   ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

icps cpsmidi
iamp ampmidi .5

kenv mxadsr 0.5, 0, 1, 0.5
asig pluck kenv, icps, icps, 2, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 2 0 4096 10 1

f0 30 ;runs 30 seconds
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*linsegr*, *expsegr*, *envlpxr*, *mxadsr*, *madsr*, *adsr*, *expon*, *expseg*, *expsega* line, *linseg*, *xtratim*

## Credits

Author: John ffitc

November 2002. Thanks to Rasmus Ekman, added documentation for the *ireltim* parameter.

November 2003. Thanks to Kanata Motohashi, fixed the link to the *linsegr* opcode.

# nchnls

nchnls — Sets the number of channels of audio output.

## Description

These statements are global value *assignments*, made at the beginning of an orchestra, before any instrument block is defined. Their function is to set certain *reserved symbol variables* that are required for performance. Once set, these reserved symbols can be used in expressions anywhere in the orchestra.

## Syntax

```
nchnls = iarg
```

## Initialization

*nchnls* = (optional) -- set number of channels of audio output to *iarg*. (1 = mono, 2 = stereo, 4 = quadraphonic.) The default value is 1 (mono).

In addition, any *global variable* can be initialized by an *init-time assignment* anywhere before the first *instr* statement. All of the above assignments are run as instrument 0 (i-pass only) at the start of real performance.

## Examples

Here is an example of the nchnls opcode. It uses the file *nchnls.csd* [examples/nchnls.csd].

### Example 592. Example of the nchnls opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac  -idac  ;;realtime audio I/O
; For Non-realtime ouput leave only the line below:
; nchnls.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2 ;two channels out
0dbfs = 1

instr 1

ainl, ainr ins ;grab your mic and sing
adel linseg 0, p3*.5, 0.02, p3*.5, 0 ;max delay time = 20ms
aoutl flanger ainl, adel, .7
aoutr flanger ainl, adel*2, .8
fout "in_s.wav", 14, aoutl, aoutr ;write to stereo file,
outs aoutl, aoutr ;16 bits with header

endin
```

```
</CsInstruments>  
<CsScore>  
  
i 1 0 10  
e  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*kr, ksmpr, sr*

## nchnls\_hw

nchnls\_hw — Returns the number of audio channels in the underlying hardware.

### Description

Returns the maximum number of audio channels in the underlying hardware. This does not necessarily correspond to the number of channels used by Csound (set by nchnls and nchnls\_i).

### Syntax

```
idacc, iadcc nchnls_hw
```

### Initialization

On init-pass idacc will contain the number of channels in the output device, and iadcc, the number of input channels. These will correspond to the currently selected/in-use hardware devices.

### Credits

Author: Victor Lazzarini  
2016

In version 6.07

# nchnls\_i

nchnls\_i — Sets the number of channels of audio input.

## Description

These statements are global value *assignments*, made at the beginning of an orchestra, before any instrument block is defined. Their function is to set certain *reserved symbol variables* that are required for performance. Once set, these reserved symbols can be used in expressions anywhere in the orchestra.

## Syntax

```
nchnls_i = iarg
```

## Initialization

*nchnls\_i* (optional) -- set number of channels of audio input to *iarg*. (1 = mono, 2 = stereo, 4 = quadraphonic.) The default value is the value of *nchnls*.

In addition, any *global variable* can be initialized by an *init-time assignment* anywhere before the first *instr statement*. All of the above assignments are run as instrument 0 (i-pass only) at the start of real performance.

## Examples

Here is an example of the nchnls\_i opcode. It uses the file *nchnls\_i.csd* [examples/nchnls\_i.csd].

### Example 593. Example of the nchnls\_i opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -idac ;;realtime audio I/O
; For Non-realtime ouput leave only the line below:
; nchnls_i.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr      = 44100
ksmps   = 32
nchnls  = 2 ;2 channels out
0dbfs   = 1
nchnls_i = 4 ;4 channels in

instr 1 ;4 channels in, two channels out

ain1, ain2, ain3, ain4 inq ;grab your mics and sing

adel1   linseg 0, p3*.5, 0.02, p3*.5, 0 ;max delay time = 20ms
adel2   linseg 0.02, p3*.5, 0, p3*.5, 0.02 ;max delay time = 20ms
aout1   flanger ain1, adel1, .7
aoutr   flanger ain2, adel*2, .8
aoutla  flanger ain3, adel2, .9
```

```

aoutra flanger ain4, adel2*2, .5
;write to quad file, 16 bits with header
fout "in_4.wav", 14, aoutl, aoutr, aoutla, aoutra
outs (aoutl+aoutla)*.5, (aoutr+aoutra)*.5 ;stereo out

endin
</CsInstruments>
<CsScore>

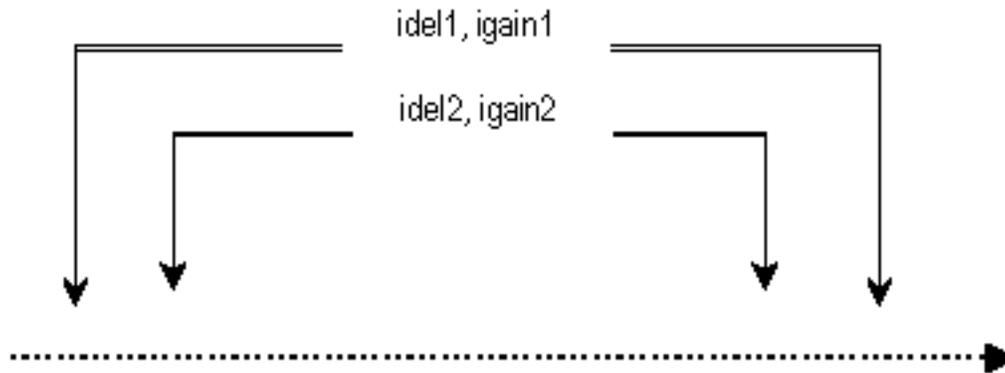
i 1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*kr, ksmips, nchnls, sr*

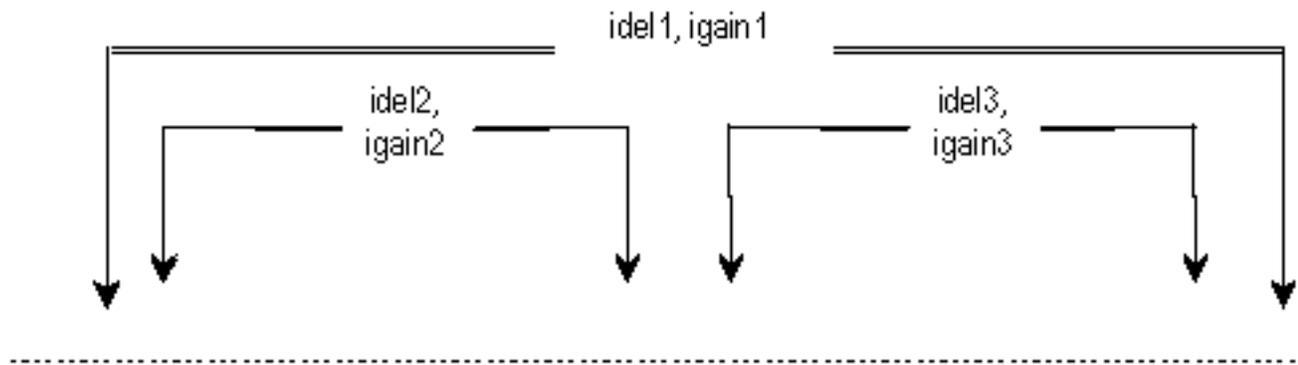


If  $imode = 2$ , the filter takes the form:



Picture of imode 2 filter.

If  $imode = 3$ , the filter takes the form:



Picture of imode 3 filter.

## Examples

Here is an example of the `nestedap` opcode. It uses the file `nestedap.csd` [examples/nestedap.csd], and `beats.wav` [examples/beats.wav].

### Example 594. Example of the `nestedap` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o nestedap.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
```

```

nchnls = 2

instr 5
  insnd = p4
  gasig diskin2 insnd, 1
endin

instr 10
  imax = 1
  idel1 = p4/1000
  igain1 = p5
  idel2 = p6/1000
  igain2 = p7
  idel3 = p8/1000
  igain3 = p9
  idel4 = p10/1000
  igain4 = p11
  idel5 = p12/1000
  igain5 = p13
  idel6 = p14/1000
  igain6 = p15

afdbk init 0

aout1 nestedap gasig+afdbk*.4, 3, imax, idel1, igain1, idel2, igain2, idel3, igain3
aout2 nestedap aout1, 2, imax, idel4, igain4, idel5, igain5
aout nestedap aout2, 1, imax, idel6, igain6

afdbk butterlp aout, 1000

outs gasig+(aout+aout1)/2, gasig-(aout+aout1)/2

gasig = 0
endin

</CsInstruments>
<CsScore>

f1 0 8192 10 1

; Diskin
; Sta Dur Soundin
i5 0 3 "beats.wav"

; Reverb
; St Dur Del1 Gn1 Del2 Gn2 Del3 Gn3 Del4 Gn4 Del5 Gn5 Del6 Gn6
i10 0 4 97 .11 23 .07 43 .09 72 .2 53 .2 119 .3
e

</CsScore>
</CsoundSynthesizer>

```

## Credits

Author: Hans Mikelson  
February 1999

New in Csound version 3.53

The example was updated May 2002, thanks to Hans Mikelson

# nlfilt

nlfilt — A filter with a non-linear effect.

## Description

Implements the filter:

$$Y\{n\} = a Y\{n-1\} + b Y\{n-2\} + d Y^2\{n-L\} + X\{n\} - C$$

described in Dobson and Fitch (ICMC'96)

## Syntax

```
ares nlfilt ain, ka, kb, kd, kC, kL
```

## Performance

1. Non-linear effect. The range of parameters are:

a = b = 0  
d = 0.8, 0.9, 0.7  
C = 0.4, 0.5, 0.6  
L = 20

This affects the lower register most but there are audible effects over the whole range. We suggest that it may be useful for coloring drums, and for adding arbitrary highlights to notes.

2. Low Pass with non-linear. The range of parameters are:

a = 0.4  
b = 0.2  
d = 0.7  
C = 0.11  
L = 20, ... 200

There are instability problems with this variant but the effect is more pronounced of the lower register, but is otherwise much like the pure comb. Short values of  $L$  can add attack to a sound.

3. High Pass with non-linear. The range of parameters are:

a = 0.35  
b = -0.3  
d = 0.95  
C = 0.2, ... 0.4

$L = 200$

4. High Pass with non-linear. The range of parameters are:

$a = 0.7$   
 $b = -0.2, \dots 0.5$   
 $d = 0.9$   
 $C = 0.12, \dots 0.24$   
 $L = 500, 10$

The high pass version is less likely to oscillate. It adds scintillation to medium-high registers. With a large delay  $L$  it is a little like a reverberation, while with small values there appear to be formant-like regions. There are arbitrary color changes and resonances as the pitch changes. Works well with individual notes.



### Warning

The "useful" ranges of parameters are not yet mapped.

## Examples

Here is an example of the `nlfilt` opcode. It uses the file `nlfilt.csd` [examples/nlfilt.csd].

### Example 595. Example of the `nlfilt` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o nlfilt.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;unfiltered noise

asig rand .7
outs asig, asig

endin

instr 2 ;filtered noise

ka = p4
kb = p5
kd = p6
kC = p7

```

```
kL = p8
asig rand .3
afilt nlfilt asig, ka, kb, kd, kC, kL
asig clip afilt, 2, .9
outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 2 ; unfiltersd

; a b d C L
i 2 2 2 0 0 0.8 0.5 20 ; non-linear effect
i 2 + 2 .4 0.2 0.7 0.11 200 ; low-pass with non-linear
i 2 + 2 0.35 -0.3 0.95 0.1 200 ; high-pass with non-linear
i 2 + 2 0.7 -0.2 0.9 0.2 20 ; high-pass with non-linear

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitth  
University of Bath/Codemist Ltd.  
Bath, UK  
1997

New in version 3.44

# nlfilt2

nlfilt2 — A filter with a non-linear effect and blowup protection.

## Description

Implements the filter:

$$Y\{n\} = \tanh(a Y\{n-1\} + b Y\{n-2\} + d Y^2\{n-L\} + X\{n\} - C)$$

described in Dobson and Fitch (ICMC'96) as modified by Risto Holopainen.

## Syntax

```
ares nlfilt2 ain, ka, kb, kd, kC, kL
```

## Performance

1. Non-linear effect. The range of parameters are:

```
a = b = 0  
d = 0.8, 0.9, 0.7  
C = 0.4, 0.5, 0.6  
L = 20
```

This affects the lower register most but there are audible effects over the whole range. We suggest that it may be useful for coloring drums, and for adding arbitrary highlights to notes.

2. Low Pass with non-linear. The range of parameters are:

```
a = 0.4  
b = 0.2  
d = 0.7  
C = 0.11  
L = 20, ... 200
```

There are instability problems with this variant but the effect is more pronounced of the lower register, but is otherwise much like the pure comb. Short values of  $L$  can add attack to a sound.

3. High Pass with non-linear. The range of parameters are:

```
a = 0.35  
b = -0.3  
d = 0.95  
C = 0.2, ... 0.4
```

$L = 200$

4. High Pass with non-linear. The range of parameters are:

a = 0.7  
b = -0.2, ... 0.5  
d = 0.9  
C = 0.12, ... 0.24  
L = 500, 10

The high pass version is less likely to oscillate. It adds scintillation to medium-high registers. With a large delay  $L$  it is a little like a reverberation, while with small values there appear to be formant-like regions. There are arbitrary color changes and resonances as the pitch changes. Works well with individual notes.



### Warning

The "useful" ranges of parameters are not yet mapped.

## Examples

Here is an example of the `nlfilt2` opcode. It uses the file `nlfilt2.csd` [examples/nlfilt2.csd].

### Example 596. Example of the `nlfilt2` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o nlfilt.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;unfiltered noise

asig rand .7
outs asig, asig

endin

instr 2 ;filtered noise

ka = p4
kb = p5
kd = p6
kC = p7
```

```
kL = p8
asig rand .3
afilt nlfilt2 asig, ka, kb, kd, kC, kL
      outs    afilt, afilt
endin
</CsInstruments>
<CsScore>

i 1 0 2    ; unfiltered

;          a    b    d    C    L
i 2 2 2 0    0    0.8  0.5  20 ; non-linear effect
i 2 + 2 .4   0.2  0.7  0.11 200 ; low=pass with non-linear
i 2 + 2 0.35 -0.3 0.95 0.1  200 ; high-pass with non-linear
i 2 + 2 0.7  -0.2 0.9  0.2  20  ; high-pass with non-linear

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitth  
University of Bath/Codemist Ltd.  
Bath, UK  
2012

New in version 5.19

# noise

noise — A white noise generator with an IIR lowpass filter.

## Description

A white noise generator with an IIR lowpass filter.

## Syntax

```
ares noise xamp, kbeta
```

## Performance

*xamp* -- amplitude of final output

*kbeta* -- beta of the lowpass filter. Should be in the range of -1 to 1, exclusive of the end-points.

The filter equation is:

$$y_n = \sqrt{(1 - \beta^2)} * x_n + \beta y_{(n-1)}$$

where  $x_n$  is the original white noise and  $y_n$  is lowpass filtered noise. The higher  $\beta$  is, the lower the filter's cut-off frequency. The cutoff frequency is roughly  $sr * ((1-kbeta)/2)$ .

## Examples

Here is an example of the noise opcode. It uses the file *noise.csd* [examples/noise.csd].

### Example 597. Example of the noise opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o noise.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
```

```

kbeta line -0.9999, p3, 0.9999 ;change beta value between -1 to 1
asig noise .3, kbeta
asig clip asig, 2, .9 ;clip signal
outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 10

e
</CsScore>
</CsoundSynthesizer>

```

Here is an example of the noise opcode controlling the kbeta parameter with a GUI interface. It uses the file *noise-2.csd* [examples/noise-2.csd].

### Example 598. Example of the noise opcode controlled with a GUI.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      ; -iadc      -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o noise.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

FLpanel "noise", 200, 50, -1, -1
    gkbeta, gslider1 FLslider "kbeta", -1, 1, 0, 5, -1, 180, 20, 10, 10
FLpanelEnd
FLrun

instr 1
    iamp = 0dbfs / 4 ; Peaks 12 dB below 0dbfs
    print iamp

    a1 noise iamp, gkbeta*0.9999
    printk2 gkbeta*0.9999
    outs a1,a1
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one minute.
i 1 0 60

e

</CsScore>
</CsoundSynthesizer>

```

## Credits

Author: John ffitc

University of Bath, Codemist. Ltd.  
Bath, UK  
December 2000

New in Csound version 4.10

# noteoff

noteoff — Send a noteoff message to the MIDI OUT port.

## Description

Send a noteoff message to the MIDI OUT port.

## Syntax

```
noteoff ichn, inum, ivel
```

## Initialization

*ichn* -- MIDI channel number (1-16)

*inum* -- note number (0-127)

*ivel* -- velocity (0-127)

## Performance

*noteon* (i-rate note on) and *noteoff* (i-rate note off) are the simplest MIDI OUT opcodes. *noteon* sends a MIDI noteon message to MIDI OUT port, and *noteoff* sends a noteoff message. A *noteon* opcode must always be followed by an *noteoff* with the same channel and number inside the same instrument, otherwise the note will play endlessly.

These *noteon* and *noteoff* opcodes are useful only when introducing a *timeout* statement to play a non-zero duration MIDI note. For most purposes, it is better to use *noteondur* and *noteondur2*.

## See Also

*noteon*, *noteondur*, *noteondur2*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# noteon

noteon — Send a noteon message to the MIDI OUT port.

## Description

Send a noteon message to the MIDI OUT port.

## Syntax

```
noteon ichn, inum, ivel
```

## Initialization

*ichn* -- MIDI channel number (1-16)

*inum* -- note number (0-127)

*ivel* -- velocity (0-127)

## Performance

*noteon* (i-rate note on) and *noteoff* (i-rate note off) are the simplest MIDI OUT opcodes. *noteon* sends a MIDI noteon message to MIDI OUT port, and *noteoff* sends a noteoff message. A *noteon* opcode must always be followed by an *noteoff* with the same channel and number inside the same instrument, otherwise the note will play endlessly.

These *noteon* and *noteoff* opcodes are useful only when introducing a *timeout* statement to play a non-zero duration MIDI note. For most purposes, it is better to use *noteondur* and *noteondur2*.

## See Also

*noteoff*, *noteondur*, *noteondur2*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# noteondur2

`noteondur2` — Sends a noteon and a noteoff MIDI message both with the same channel, number and velocity.

## Description

Sends a noteon and a noteoff MIDI message both with the same channel, number and velocity.

## Syntax

```
noteondur2 ichn, inum, ivel, idur
```

## Initialization

`ichn` -- MIDI channel number (1-16)

`inum` -- note number (0-127)

`ivel` -- velocity (0-127)

`idur` -- how long, in seconds, this note should last.

## Performance

`noteondur2` (i-rate note on with duration) sends a noteon and a noteoff MIDI message both with the same channel, number and velocity. Noteoff message is sent after `idur` seconds are elapsed by the time `noteondur2` was active.

`noteondur` differs from `noteondur2` in that `noteondur` truncates note duration when current instrument is deactivated by score or by real-time playing, while `noteondur2` will extend performance time of current instrument until `idur` seconds have elapsed. In real-time playing, it is suggested to use `noteondur` also for undefined durations, giving a large `idur` value.

Any number of `noteondur2` opcodes can appear in the same Csound instrument, allowing chords to be played by a single instrument.

## Examples

Here is an example of the `noteondur2` opcode. It uses the file `noteondur2.csd` [examples/noteondur2.csd].

### Example 599. Example of the `noteondur2` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

This example generates notes for every note received on the MIDI input. It generates MIDI notes on csound's MIDI output, so be sure to connect something.

```
<CsoundSynthesizer>  
<CsOptions>
```

```
; Select audio/midi flags here according to platform
; Audio out   Audio in   No messages
-odac        -iadc       -d           -M0  -Q1;;;RT audio I/O with MIDI in
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

; Example by Giorgio Zucco 2007

instr 1

  ifund  notnum
  ivel  veloc
  idur = 1

  ;chord with single key
  noteondur2 1, ifund,  ivel, idur
  noteondur2 1, ifund+3, ivel, idur
  noteondur2 1, ifund+7, ivel, idur
  noteondur2 1, ifund+9, ivel, idur

endin

</CsInstruments>
<CsScore>
; Dummy ftable
f 0 60
</CsScore>
</CsoundSynthesizer>
```

## See Also

*noteoff, noteon, noteondur, midion, midion2*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# noteondur

`noteondur` — Sends a noteon and a noteoff MIDI message both with the same channel, number and velocity.

## Description

Sends a noteon and a noteoff MIDI message both with the same channel, number and velocity.

## Syntax

```
noteondur ichn, inum, ivel, idur
```

## Initialization

`ichn` -- MIDI channel number (1-16)

`inum` -- note number (0-127)

`ivel` -- velocity (0-127)

`idur` -- how long, in seconds, this note should last.

## Performance

`noteondur` (i-rate note on with duration) sends a noteon and a noteoff MIDI message both with the same channel, number and velocity. Noteoff message is sent after `idur` seconds are elapsed by the time `noteondur` was active.

`noteondur` differs from `noteondur2` in that `noteondur` truncates note duration when current instrument is deactivated by score or by real-time playing, while `noteondur2` will extend performance time of current instrument until `idur` seconds have elapsed. In real-time playing, it is suggested to use `noteondur` also for undefined durations, giving a large `idur` value.

Any number of `noteondur` opcodes can appear in the same Csound instrument, allowing chords to be played by a single instrument.

## Examples

Here is an example of the `noteondur` opcode. It uses the file `noteondur.csd` [examples/noteondur.csd].

### Example 600. Example of the `noteondur` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

This example generates notes for every note received on the MIDI input. It generates MIDI notes on csound's MIDI output, so be sure to connect something.

```
<CsoundSynthesizer>  
<CsOptions>
```

```
; Select audio/midi flags here according to platform
; Audio out   Audio in   No messages
-odac         -iadc      -d           -M0  -Q1;;;RT audio I/O with MIDI in
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

; Example by Giorgio Zucco 2007

instr 1  ;Turned on by MIDI notes on channel 1

  ifund  notnum
  ivel  veloc
  idur = 1

  ;chord with single key
  noteondur 1, ifund,  ivel, idur
  noteondur 1, ifund+3, ivel, idur
  noteondur 1, ifund+7, ivel, idur
  noteondur 1, ifund+9, ivel, idur

endin

</CsInstruments>
<CsScore>
; Play Instrument #1 for 60 seconds.

i1 0 60

</CsScore>
</CsoundSynthesizer>
```

## See Also

*noteoff, noteon, noteondur2, midion, midion2*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# notnum

notnum — Get a note number from a MIDI event.

## Description

Get a note number from a MIDI event.

## Syntax

```
ival notnum
```

## Performance

Get the MIDI byte value (0 - 127) denoting the note number of the current event.

## Examples

Here is an example of the notnum opcode. It uses the file *notnum.csd* [examples/notnum.csd].

### Example 601. Example of the notnum opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -M0 --rtmidi=virtual ;;realtime audio out with virtual MIDI in
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

massign 1, 1 ; set MIDI channel 1 to play instr 1

instr 1

iNum notnum
  print iNum
; Convert MIDI note number to Hz
iHz = (440.0*exp(log(2.0)*((iNum)-69.0)/12.0))
aosc oscil 0.6, iHz, 1
  outs aosc, aosc

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine wave

f 0 60 ;play 60 seconds

e
```

```
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*aftouch, ampmidi, cpsmidi, cpsmidib, midictrl, octmidi, octmidib, pchbend, pchmidi, pchmidib, veloc*

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

Example written by David Akbari.

# nreverb

nreverb — A reverberator consisting of 6 parallel comb-lowpass filters.

## Description

This is a reverberator consisting of 6 parallel comb-lowpass filters being fed into a series of 5 allpass filters. *nreverb* replaces *reverb2* (version 3.48) and so both opcodes are identical.

## Syntax

```
ares nreverb asig, ktime, khdif [, iskip] [,inumCombs] [, ifnCombs] \  
      [, inumAlpas] [, ifnAlpas]
```

## Initialization

*iskip* (optional, default=0) -- Skip initialization if present and non-zero.

*inumCombs* (optional) -- number of filter constants in comb filter. If omitted, the values default to the nreverb constants. New in Csound version 4.09.

*ifnCombs* - function table with *inumCombs* comb filter time values, followed the same number of gain values. The ftable should not be rescaled (use negative fgen number). Positive time values are in seconds. The time values are converted internally into number of samples, then set to the next greater prime number. If the time is negative, it is interpreted directly as time in sample frames, and no processing is done (except negation). New in Csound version 4.09.

*inumAlpas*, *ifnAlpas* (optional) -- same as *inumCombs/ifnCombs*, for allpass filter. New in Csound 4.09.

## Performance

The input signal *asig* is reverberated for *ktime* seconds. The parameter *khdif* controls the high frequency diffusion amount. The values of *khdif* should be from 0 to 1. If *khdif* is set to 0 the all the frequencies decay with the same speed. If *khdif* is 1, high frequencies decay faster than lower ones. If *ktime* is inadvertently set to a non-positive number, *ktime* will be reset automatically to 0.01. (New in Csound version 4.07.)

As of Csound version 4.09, *nreverb* may read any number of comb and allpass filter from an ftable.

## Examples

Here is a simple example of the nreverb opcode. It uses the file *nreverb.csd* [examples/nreverb.csd].

### Example 602. Simple example of the nreverb opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
; Audio out  Audio in  No messages
```

```

-odac          -iadc    -d      ;;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o nreverb.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

gaout init 0

instr 1
a1 oscil 15000, 440, 1
out a1

gaout = gaout+a1
endin

instr 99

a2 nreverb gaout, 2, .3
out a2*.15 ;volume of reverb

gaout = 0
endin

</CsInstruments>
<CsScore>

; Table 1: an ordinary sine wave.
f 1 0 32768 10 1

i 1 0 .5
i 1 1 .5
i 1 2 .5
i 1 3 .5
i 1 4 .5
i 99 0 9
e

</CsScore>
</CsoundSynthesizer>

```

Here is an example of the `nreverb` opcode using an `ftable` for filter constants. It uses the file `nreverb_ftable.csd` [examples/nreverb\_ftable.csd], and `beats.wav` [examples/beats.wav].

### Example 603. An example of the `nreverb` opcode using an `ftable` for filter constants.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc    -d      ;;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o nreverb_ftable.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410

```

```
ksmps = 10
nchnls = 1

instr 1
  a1 soundin "beats.wav"
  a2 nreverb a1, 1.5, .75, 0, 8, 71, 4, 72
  out a1 + a2 * .4
endin

</CsInstruments>
<CsScore>

; freeverb time constants, as direct (negative) sample, with arbitrary gains
f71 0 16 -2 -1116 -1188 -1277 -1356 -1422 -1491 -1557 -1617 0.8 0.79 0.78 0.77 0.76 0.75 0.74

f72 0 16 -2 -556 -441 -341 -225 0.7 0.72 0.74 0.76

i1 0 3
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Authors: Paris Smaragdis (*reverb2*)  
MIT, Cambridge  
1995

Author: Richard Karpen (*nreverb*)  
Seattle, Wash  
1998

# nrpn

nrpn — Sends a Non-Registered Parameter Number to the MIDI OUT port.

## Description

Sends a NPRN (Non-Registered Parameter Number) message to the MIDI OUT port each time one of the input arguments changes.

## Syntax

```
nrpn kchan, kparmnum, kparmvalue
```

## Performance

*kchan* -- MIDI channel (1-16)

*kparmnum* -- number of NRPN parameter

*kparmvalue* -- value of NRPN parameter

This opcode sends new message when the MIDI translated value of one of the input arguments changes. It operates at k-rate. Useful with the MIDI instruments that recognize NRPNs (for example with the newest sound-cards with internal MIDI synthesizer such as SB AWE32, AWE64, GUS etc. in which each patch parameter can be changed during the performance via NRPN)

## Examples

Here is an example of the nrpn opcode. It uses the file *nrpn.csd* [examples/nrpn.csd].

### Example 604. Example of the nrpn opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

This example generates notes for every note received on the MIDI input. It generates MIDI notes on csound's MIDI output, so be sure to connect something.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -Q1   ;;;realtime audio out with MIDI out
;-iadc     ;;;uncomment -iadc if realtime audio input is needed too
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; change attack time of external synth

initc7 1, 6, 0 ; set controller 6 to 0
```

```
nrpn 1, 99, 1 ; set MSB
nrpn 1, 98, 99 ; set LSB
katt ctrl17 1, 6, 1, 127 ; DataEntMSB
idur = 2
noteondur2 1, 60, 100, idur ; play note on synth

endin
</CsInstruments>
<CsScore>

i 1 0 3
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Gabriel Maldonado  
Italy  
1998

New in Csound version 3.492

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# nsamp

nsamp — Returns the number of samples loaded into a stored function table number.

## Description

Returns the number of samples loaded into a stored function table number.

## Syntax

```
nsamp(x) (init-rate args only)
```

## Performance

Returns the number of samples loaded into stored function table number *x* by GEN01. This is useful when a sample is shorter than the power-of-two function table that holds it. New in Csound version 3.49.

As of Csound version 5.02, *nsamp* works with deferred-length function tables (see *GEN01*).

*nsamp* differs from *flen* in that *nsamp* gives the number of sample frames loaded, while *flen* gives the total number of samples. For example, with a stereo sound file of 10000 samples, *flen*() would return 19999 (i.e. a total of 20000 mono samples, not including a guard point), but *nsamp*() returns 10000.

## Examples

Here is an example of the *nsamp* opcode. It uses the file *nsamp.csd* [examples/nsamp.csd], *kickroll.wav* [examples/kickroll.wav], and *fox.wav* [examples/fox.wav].

### Example 605. Example of the nsamp opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o nsamp.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
; slightly adapted example from Jonathan Murphy Dec 2006

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifn = p4 ; table number
ilen = nsamp(ifn)
prints "actual numbers of samples = %d\n", ilen
itrns = 1 ; no transposition
```

```
ilps = 0 ; loop starts at index 0
ilpe = ilen ; ends at value returned by nsamp above
imode = 1 ; loops forward
istrt = 0 ; commence playback at index 0 samples
; lphasor provides index into f1
alphs lphasor itrns, ilps, ilpe, imode, istrt
atab tablei alphs, ifn
outs atab, atab

endin
</CsInstruments>
<CsScore>
f 1 0 262144 1 "kickroll.wav" 0 4 1 ;stereo file in table, with lots of zeroes
f 2 0 262144 1 "fox.wav" 0 4 1 ;mono file in table, with lots of zeroes

i1 0 10 1
i1 + 10 2
e

</CsScore>
</CsoundSynthesizer>
```

Since the stereo audio file “kickroll.wav” has 37792 samples and the mono file “fox.wav” has 121569 samples, its output should include lines like these:

```
actual numbers of samples = 37792
actual numbers of samples = 121569
```

## See Also

*ftchnls, filen, flptim, ftsr*

## Credits

Author: Gabriel Maldonado  
Italy  
October 1998

## nstance

`nstance` — Schedules a new instrument instance, storing the instance handle in a variable.

### Description

Schedules a new instrument `nstance`, returning a handle that can be used later to refer directly to the running `nstance`. This opcode is similar to `schedule`, but has the added facility of retrieving the `nstance` handle.

### Syntax

```
iHandle nstance insnum, iwhen, idur [, ip4] [, ip5] [...]
```

```
iHandle nstance "insname", iwhen, idur [, ip4] [, ip5] [...]
```

### Initialization

*iHandle* -- this variable will contain a handle to the event instance. It can be used, for example, to stop the given instrument instance via the `turnoff` opcode. Note that the instance handle is only valid once the instrument is initialised.

*insnum* -- instrument number. Equivalent to `p1` in a score *i statement*. *insnum* must be a number greater than the number of the calling instrument.

*"insname"* -- A string (in double-quotes) representing a named instrument.

*iwhen* -- start time of the new event. Equivalent to `p2` in a score *i statement*. *iwhen* must be nonnegative. If *iwhen* is zero, *insnum* must be greater than or equal to the `p1` of the current instrument.

*idur* -- duration of event. Equivalent to `p3` in a score *i statement*.

*ip4, ip5, ...* -- Equivalent to `p4, p5, etc.`, in a score *i statement*.

### Performance

`nstance` adds a new score event. The arguments, including options, are the same as in a score. The *iwhen* time (`p2`) is measured from the time of this event.

If the duration is zero or negative the new event is of MIDI type, and inherits the release sub-event from the scheduling instruction.



#### Note

Note that the `nstance` opcode can't accept string p-fields. If you need to pass strings when instantiating an instrument, use the `scoreline` or `scoreline_i` opcode.

### Examples

Here is an example of the `nstance` opcode. It uses the file `nstance.csd` [examples/nstance.csd].

## Example 606. Example of the nstance opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o nstance.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - oscillator with a high note.
instr 1
; Play Instrument #2 at the same time.
iHandle nstance 2, 0, p3

; Play a high note.
a1 oscils 10000, 880, 1
out a1
endin

; Instrument #2 - oscillator with a low note.
instr 2
; Play a low note.
a1 oscils 10000, 220, 1
out a1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for half a second.
i 1 0 0.5
; Play Instrument #1 for half a second.
i 1 1 0.5
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*schedwhen*

## See also

*event, event\_i, schedwhen, schedkwhen, schedkwhennamed, scoreline, scoreline\_i*

More information on this opcode: [http://www.csoundjournal.com/issue15/phrase\\_loops.html](http://www.csoundjournal.com/issue15/phrase_loops.html) , written by Jim Aikin

## Credits

Author: Victor Lazzarini  
November 2013

# ntom

ntom — Convert note name to midi note number

## Description

Convert note name to midi note number. It allows note name to include microtones or a deviation in cents.

## Syntax

```
kmidi ntom Snote
```

```
imidi ntom Snote
```

## Performance

*Snote* -- Note name

*kmidi* -- Midi note number

Example of note names:

midi	note name
60	4C
60.4	4C+40
60.5	4C+
60.9	4Db-10
61	4C#
61.5	4D-



### Note

4C is the central C in the piano

## Examples

Here is an example of the ntom opcode. It uses the file *mton-ntom.csd* [examples/mton-ntom.csd].

### Example 607. Example of the fton opcode.

```
<CsoundSynthesizer>
<CsOptions>
--nosound
</CsOptions>
<CsInstruments>

instr 1
  S4 mton ntom( "7D+63" )
  puts S4, 1

  S1 mton 60
```

```
puts S1, 1

S2 mton fton(442)
puts S2, 1

S3 = mton(48.25)
puts S3, 1

k1 = ntom("4C")
printk2 k1

i2 ntom "4E"
print i2

S5 = mton(ntom("4G+"))
puts S5, 1

turnoff
endin

instr 2
k1 = ntom("4Eb-31")
printk2 k1
i0 ntom "4C+"
print i0
i1 = ntom:i("4A")
print i1
turnoff
endin

</CsInstruments>
<CsScore>

i 1 0 1
i 2 0 1

</CsScore>
</CsoundSynthesizer>
```

## See Also

*cpsmidim, mtof, fton, mton*

## Credits

By: Eduardo Moguillansky 2017

New in version 6.09

## nstrnum

nstrnum — Returns the number of a named instrument.

### Description

Returns the number of a named instrument.

### Syntax

```
insno nstrnum "name"
```

### Initialization

*insno* -- the instrument number of the named instrument.

### Performance

*"name"* -- the named instrument's name.

If an instrument with the specified name does not exist, an init error occurs, and -1 is returned.

### Credits

Author: Istvan Varga  
New in version 4.23  
Written in the year 2002.

# ntrpol

ntrpol — Calculates the weighted mean value of two input signals.

## Description

Calculates the weighted mean value (i.e. linear interpolation) of two input signals

## Syntax

```
ares ntrpol asig1, asig2, kpoint [, imin] [, imax]
ires ntrpol isig1, isig2, ipoint [, imin] [, imax]
kres ntrpol ksig1, ksig2, kpoint [, imin] [, imax]
```

## Initialization

*imin* -- minimum xpoint value (optional, default 0)

*imax* -- maximum xpoint value (optional, default 1)

## Performance

*xsig1*, *xsig2* -- input signals

*xpoint* -- interpolation point between the two values

*ntrpol* opcode outputs the linear interpolation between two input values. *xpoint* is the distance of evaluation point from the first value. With the default values of *imin* and *imax*, (0 and 1) a zero value indicates no distance from the first value and the maximum distance from the second one. With a 0.5 value, *ntrpol* will output the mean value of the two inputs, indicating the exact half point between *xsig1* and *xsig2*. A 1 value indicates the maximum distance from the first value and no distance from the second one. The range of *xpoint* can be also defined with *imin* and *imax* to make its management easier.

These opcodes are useful for crossfading two signals.

## Examples

Here is an example of the *ntrpol* opcode. It uses the file *ntrpol.csd* [examples/ntrpol.csd].

### Example 608. Example of the *ntrpol* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if real audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o ntrpol.wav -W ;; for file output any platform
```

```
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSin ftgen 1, 0, 1024, 10, 1

instr 1

avco vco2 .5, 110 ;sawtooth wave
asin poscil .5, 220, giSin ;sine wave but octave higher
kx linseg 0, p3*.4, 1, p3*.6, 1 ;crossfade between saw and sine
asig ntrpol avco, asin, kx
outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 5
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Gabriel Maldonado

Italy

October 1998

New in Csound version 3.49

# octave

octave — Calculates a factor to raise/lower a frequency by a given amount of octaves.

## Description

Calculates a factor to raise/lower a frequency by a given amount of octaves.

## Syntax

```
octave(x)
```

This function works at a-rate, i-rate, and k-rate.

## Initialization

*x* -- a value expressed in octaves.

## Performance

The value returned by the *octave* function is a factor. You can multiply a frequency by this factor to raise/lower it by the given amount of octaves.

## Examples

Here is an example of the octave opcode. It uses the file *octave.csd* [examples/octave.csd].

### Example 609. Example of the octave opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o octave.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

iroot = 440 ; root note is A above middle-C (440 Hz)
koc = lfo 5, 1, 5 ; generate sawtooth, go from 5 octaves higher to root
koc = int(koc) ; produce only whole numbers
kfactor = octave(koc) ; for octave
knew = iroot * kfactor
```

```
printk2 knew

asig pluck 1, knew, 1000, 0, 1
asig dcblock asig ;remove DC
outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 5
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like:

```
i1 3520.00000
i1 1760.00000
i1 880.00000
i1 440.00000
i1 7040.00000
i1 3520.00000
i1 1760.00000
i1 880.00000
i1 440.00000
.....
```

## See Also

*cent, db, semitone*

## Credits

New in version 4.16

# octcps

octcps — Converts a cycles-per-second value to octave-point-decimal.

## Description

Converts a cycles-per-second value to octave-point-decimal.

## Syntax

```
octcps (cps) (init- or control-rate args only)
```

where the argument within the parentheses may be a further expression.

## Performance

*octcps* and its related opcodes are really *value converters* with a special function of manipulating pitch data.

Data concerning pitch and frequency can exist in any of the following forms:

**Table 16. Pitch and Frequency Values**

Name	Abbreviation
octave point pitch-class (8ve.pc)	pch
octave point decimal	oct
cycles per second	cps
Midi note number (0-127)	midinn

The first two forms consist of a whole number, representing octave registration, followed by a specially interpreted fractional part. For *pch*, the fraction is read as two decimal digits representing the 12 equal-tempered pitch classes from .00 for C to .11 for B. For *oct*, the fraction is interpreted as a true decimal fractional part of an octave. The two fractional forms are thus related by the factor 100/12. In both forms, the fraction is preceded by a whole number octave index such that 8.00 represents Middle C, 9.00 the C above, etc. Midi note number values range between 0 and 127 (inclusively) with 60 representing Middle C, and are usually whole numbers. Thus A440 can be represented alternatively by 440 (*cps*), 69 (*midinn*), 8.09 (*pch*), or 8.75 (*oct*). Microtonal divisions of the *pch* semitone can be encoded by using more than two decimal places.

The mnemonics of the pitch conversion units are derived from morphemes of the forms involved, the second morpheme describing the source and the first morpheme the object (result). Thus *cpspch*(8.09) will convert the pitch argument 8.09 to its *cps* (or Hertz) equivalent, giving the value of 440. Since the argument is constant over the duration of the note, this conversion will take place at *i*-time, before any samples for the current note are produced.

By contrast, the conversion *cpsoct*(8.75 + *k1*) which gives the value of A440 transposed by the octave interval *k1*. The calculation will be repeated every *k*-period since that is the rate at which *k1* varies.



### Note

The conversion from *pch*, *oct*, or *midinn* into *cps* is not a linear operation but involves an exponential process that could be time-consuming when executed repeatedly. Csound now

uses a built-in table lookup to do this efficiently, even at audio rates. Because the table index is truncated without interpolation, pitch resolution when using one of these opcodes is limited to 8192 discrete and equal divisions of the octave, and some pitches of the standard 12-tone equally-tempered scale are very slightly mistuned (by at most 0.15 cents).

## Examples

Here is an example of the `octcps` opcode. It uses the file `octcps.csd` [examples/octcps.csd].

### Example 610. Example of the `octcps` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o octcps.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Convert a cycles-per-second value into an
; octave value.
icps = 440
ioct = octcps(icps)

print ioct
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include a line like this:

```
instr 1:  ioct = 8.750
```

## See Also

*cpsoct*, *cpspch*, *octpch*, *pchoct*, *cpsmidinn*, *octmidinn*, *pchmidinn*

## Credits

Example written by Kevin Conder.

# octmidi

octmidi — Get the note number, in octave-point-decimal units, of the current MIDI event.

## Description

Get the note number, in octave-point-decimal units, of the current MIDI event.

## Syntax

```
iout octmidi
```

## Performance

Get the note number of the current MIDI event, expressed in octave-point-decimal units, for local processing.



### octmidi vs. octmidinn

The *octmidi* opcode only produces meaningful results in a Midi-activated note (either real-time or from a Midi score with the -F flag). With *octmidi*, the Midi note number value is taken from the Midi event that is internally associated with the instrument instance. On the other hand, the *octmidinn* opcode may be used in any Csound instrument instance whether it is activated from a Midi event, score event, line event, or from another instrument. The input value for *octmidinn* might for example come from a p-field in a textual score or it may have been retrieved from the real-time Midi event that activated the current note using the *notnum* opcode.

## Examples

Here is an example of the octmidi opcode. It uses the file *octmidi.csd* [examples/octmidi.csd].

### Example 611. Example of the octmidi opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages  MIDI in
-odac          -iadc      -d          -M0    ;;RT audio I/O with MIDI in
; For Non-realtime ouput leave only the line below:
; -o octmidi.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1
```

```
; Instrument #1.
instr 1
  ; This example expects MIDI note inputs on channel 1
  i1 octmidi

  print i1
endin

</CsInstruments>
<CsScore>

;Dummy f-table to give time for real-time MIDI events
f 0 8000
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*aftouch, ampmidi, cpsmidi, cpsmidib, midictrl, notnum, octmidib, pchbend, pchmidi, pchmidib, veloc, cpsmidinn, octmidinn, pchmidinn*

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

Example written by Kevin Conder.

# octmidib

octmidib — Get the note number of the current MIDI event and modify it by the current pitch-bend value, express it in octave-point-decimal.

## Description

Get the note number of the current MIDI event and modify it by the current pitch-bend value, express it in octave-point-decimal.

## Syntax

```
iOct octmidib [irange]
kOct octmidib [irange]
```

## Initialization

*irange* (optional) -- the pitch bend range in semitones

## Performance

Get the note number of the current MIDI event, modify it by the current pitch-bend value, and express the result in octave-point-decimal units. Available as an i-time value or as a continuous k-rate value.

## Examples

Here is an example of the octmidib opcode. It uses the file *octmidib.csd* [examples/octmidib.csd].

### Example 612. Example of the octmidib opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages  MIDI in
-odac          -iadc      -d           -M0   ;;RT audio I/O with MIDI in
; For Non-realtime ouput leave only the line below:
; -o octmidib.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; This example expects MIDI note inputs on channel 1
i1 octmidib
```

```
    print i1
  endin

  </CsInstruments>
  <CsScore>

  ;Dummy f-table to give time for real-time MIDI events
  f 0 8000
  e

  </CsScore>
</CsoundSynthesizer>
```

## See Also

*aftouch, ampmidi, cpsmidi, cpsmidib, midictrl, notnum, octmidi, pchbend, pchmidi, pchmidib, veloc*

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

Example written by Kevin Conder.

# octmidinn

octmidinn — Converts a Midi note number value to octave-point-decimal.

## Description

Converts a Midi note number value to octave-point-decimal.

## Syntax

```
octmidinn (MidiNoteNumber) (init- or control-rate args only)
```

where the argument within the parentheses may be a further expression.

## Performance

*octmidinn* is a function that takes an i-rate or k-rate value representing a Midi note number and returns the equivalent pitch value in Csound's octave-point-decimal format. This conversion assumes that Middle C (8.000 in *oct*) is Midi note number 60. Midi note number values are typically integers in the range from 0 to 127 but fractional values or values outside of this range will be interpreted consistently.



### octmidinn vs. octmidi

The *octmidinn* opcode may be used in any Csound instrument instance whether it is activated from a Midi event, score event, line event, or from another instrument. The input value for *octmidinn* might for example come from a p-field in a textual score or it may have been retrieved from the real-time Midi event that activated the current note using the *notnum* opcode. You must specify an i-rate or k-rate expression for the Midi note number that is to be converted. On the other hand, the *octmidi* opcode only produces meaningful results in a Midi-activated note (either real-time or from a Midi score with the -F flag). With *octmidi*, the Midi note number value is taken from the Midi event associated with the instrument instance, and no location or expression for this value may be specified.

*octmidinn* and its related opcodes are really *value converters* with a special function of manipulating pitch data.

Data concerning pitch and frequency can exist in any of the following forms:

**Table 17. Pitch and Frequency Values**

Name	Abbreviation
octave point pitch-class (8ve.pc)	pch
octave point decimal	oct
cycles per second	cps
Midi note number (0-127)	midinn

The first two forms consist of a whole number, representing octave registration, followed by a specially interpreted fractional part. For *pch*, the fraction is read as two decimal digits representing the 12 equal-tempered pitch classes from .00 for C to .11 for B. For *oct*, the fraction is interpreted as a true decimal fractional part of an octave. The two fractional forms are thus related by the factor 100/12. In both forms,

the fraction is preceded by a whole number octave index such that 8.00 represents Middle C, 9.00 the C above, etc. Midi note number values range between 0 and 127 (inclusively) with 60 representing Middle C, and are usually whole numbers. Thus A440 can be represented alternatively by 440 (*cps*), 69 (*midinn*), 8.09 (*pch*), or 8.75 (*oct*). Microtonal divisions of the *pch* semitone can be encoded by using more than two decimal places.

The mnemonics of the pitch conversion units are derived from morphemes of the forms involved, the second morpheme describing the source and the first morpheme the object (result). Thus *cpspch*(8.09) will convert the pitch argument 8.09 to its *cps* (or Hertz) equivalent, giving the value of 440. Since the argument is constant over the duration of the note, this conversion will take place at i-time, before any samples for the current note are produced.

By contrast, the conversion *cpsoct*(8.75 + k1) which gives the value of A440 transposed by the octave interval *k1*. The calculation will be repeated every *k*-period since that is the rate at which *k1* varies.



## Note

The conversion from *pch*, *oct*, or *midinn* into *cps* is not a linear operation but involves an exponential process that could be time-consuming when executed repeatedly. Csound now uses a built-in table lookup to do this efficiently, even at audio rates. Because the table index is truncated without interpolation, pitch resolution when using one of these opcodes is limited to 8192 discrete and equal divisions of the octave, and some pitches of the standard 12-tone equally-tempered scale are very slightly mistuned (by at most 0.15 cents).

## Examples

Here is an example of the *octmidinn* opcode. It uses the file *cpsmidinn.csd* [examples/cpsmidinn.csd].

### Example 613. Example of the *octmidinn* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
; Prints a table showing the equivalents of all Midi
; note numbers from 0-127 in cycles-per-second,
; octave.decimal, and octave.pitchclass units.

<CsOptions>
; Select audio/midi flags here according to platform.
; This example produces no audio, so we render in
; non-realttime and turn off sound to disk:
-n
</CsOptions>
<CsInstruments>

instr 1
; i-time loop to print conversion table
imidiNN = 0
loop1:
icps = cpsmidinn(imidiNN)
ioct = octmidinn(imidiNN)
ipch = pchmidinn(imidiNN)

print imidiNN, icps, ioct, ipch

imidiNN = imidiNN + 1
if (imidiNN < 128) igoto loop1
```

```
    endin

    instr 2
      ; test k-rate converters
      kMiddleC = 60
      kcps = cpsmidinn(kMiddleC)
      koct = octmidinn(kMiddleC)
      kpch = pchmidinn(kMiddleC)

      printks "%d %f %f %f\n", 1.0, kMiddleC, kcps, koct, kpch
    endin

    </CsInstruments>
    <CsScore>
    i1 0 0
    i2 0 0.1
    e

    </CsScore>
  </CsoundSynthesizer>
```

## See Also

*cpsmidinn, pchmidinn, octmidi, notnum, cpspch, cpsoct, octcps, octpch, pchoct*

## Credits

Derived from original value converters by Barry Vercoe.

New in version 5.07

# octpch

octpch — Converts a pitch-class value to octave-point-decimal.

## Description

Converts a pitch-class value to octave-point-decimal.

## Syntax

`octpch (pch) (init- or control-rate args only)`

where the argument within the parentheses may be a further expression.

## Performance

*octpch* and its related opcodes are really *value converters* with a special function of manipulating pitch data.

Data concerning pitch and frequency can exist in any of the following forms:

**Table 18. Pitch and Frequency Values**

Name	Abbreviation
octave point pitch-class (8ve.pc)	pch
octave point decimal	oct
cycles per second	cps
Midi note number (0-127)	midinn

The first two forms consist of a whole number, representing octave registration, followed by a specially interpreted fractional part. For *pch*, the fraction is read as two decimal digits representing the 12 equal-tempered pitch classes from .00 for C to .11 for B. For *oct*, the fraction is interpreted as a true decimal fractional part of an octave. The two fractional forms are thus related by the factor 100/12. In both forms, the fraction is preceded by a whole number octave index such that 8.00 represents Middle C, 9.00 the C above, etc. Midi note number values range between 0 and 127 (inclusively) with 60 representing Middle C, and are usually whole numbers. Thus A440 can be represented alternatively by 440 (*cps*), 69 (*midinn*), 8.09 (*pch*), or 8.75 (*oct*). Microtonal divisions of the *pch* semitone can be encoded by using more than two decimal places.

The mnemonics of the pitch conversion units are derived from morphemes of the forms involved, the second morpheme describing the source and the first morpheme the object (result). Thus *cpspch*(8.09) will convert the pitch argument 8.09 to its *cps* (or Hertz) equivalent, giving the value of 440. Since the argument is constant over the duration of the note, this conversion will take place at i-time, before any samples for the current note are produced.

By contrast, the conversion *cpsoct*(8.75 + k1) which gives the value of A440 transposed by the octave interval *k1*. The calculation will be repeated every k-period since that is the rate at which *k1* varies.



### Note

The conversion from *pch*, *oct*, or *midinn* into *cps* is not a linear operation but involves an exponential process that could be time-consuming when executed repeatedly. Csound now

uses a built-in table lookup to do this efficiently, even at audio rates. Because the table index is truncated without interpolation, pitch resolution when using one of these opcodes is limited to 8192 discrete and equal divisions of the octave, and some pitches of the standard 12-tone equally-tempered scale are very slightly mistuned (by at most 0.15 cents).

## Examples

Here is an example of the `octpch` opcode. It uses the file `octpch.csd` [examples/octpch.csd].

### Example 614. Example of the `octpch` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o octpch.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Convert a pitch-class value into an
; octave-point-decimal value.
ipch = 8.09
ioct = octpch(ipch)

print ioct
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include a line like this:

```
instr 1:  ioct = 8.750
```

## See Also

*cpsoct*, *cpspch*, *octcps*, *pchoct*, *cpsmidinn*, *octmidinn*, *pchmidinn*

## Credits

Example written by Kevin Conder.

# olabuffer

olabuffer — Sum overlapping frames of audio as k-rate arrays and read as an audio signal

## Description

*olabuffer* This opcode takes 1 dimensional k-rate arrays that contain sequential frames of audio and sums them based on an overlap factor resulting in an audio signal output. This is useful for frame based audio processing such as spectral analysis/synthesis.

## Syntax

```
aout olabuffer kin, ioverlap
```

## Initialization

*ioverlap* -- The amount of overlaps per k-array input frame sample size. For example for an input window size of 1024, and a hop size of 256 the overlap factor would be 4. The overlap factor must be larger than or equal to *ksmps* and must also be an integer multiple of the input k-rate array sample count.

## Performance

*aout* -- The resulting audio signal from the added input frames. *kin* -- A k-rate array containing sequential frames of audio.

## Examples

Here is a simple example of the *olabuffer* opcode. It uses the file *framebuffer.csd* [examples/framebuffer.csd].

### Example 615. Example of the *olabuffer* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
nchnls = 2
0dbfs = 1
ksmps = 128
sr = 44100

instr 1
  isize init 1024
  ioverlaps init 4

  asig diskin2 "fox.wav", 1, 0, 1
  kframe[] framebuffer asig, isize
  kwindowedFrame[] window kframe, isize
```

```
    aout olabuffer kwindowedFrame, ioverlaps
    aout = aout / 2

    outs aout, aout
endin

</CsInstruments>
<CsScore>
i 1 0 400
</CsScore>
</CsoundSynthesizer>
```

## See Also

*olabuffer shiftin shiftout*

## Credits

Author: Edward Costello;  
NUIM, 2015

New in version 6.06

# opcode

opcode — Defines the start of user-defined opcode block.

## Defining opcodes

The *opcode* and *endop* statements allow defining a new opcode that can be used the same way as any of the built-in Csound opcodes. These opcode blocks are very similar to instruments (and are, in fact, implemented as special instruments), but cannot be called as a normal instrument e.g. with the *i* statements.

A user-defined opcode block must precede the instrument (or other opcode) from which it is used. But it is possible to call the opcode from itself. This allows recursion of any depth that is limited only by available memory. Additionally, there is an experimental feature that allows running the opcode definition at a higher control rate than the *kr* value specified in the orchestra header.

Similarly to instruments, the variables and labels of a user-defined opcode block are local and cannot be accessed from the caller instrument (and the opcode cannot access variables of the caller, either).

Some parameters are automatically copied at initialization, however:

- all p-fields up to and including the highest-numbered one referenced in the calling instrument
- extra time (see also *xtratim*, *linsegr*, and related opcodes). This may affect the operation of *linsegr/ex-psegr/linenr/envlpxr* in the user-defined opcode block.
- MIDI parameters, if there are any.

Also, the release flag (see the *release* opcode) is copied at performance time.

Modifying the note duration in the opcode definition by assigning to *p3*, or using *ihold*, *turnoff*, *xtratim*, *linsegr*, or similar opcodes will also affect the caller instrument. Changes to MIDI controllers (for example with *ctrlinit*) will also apply to the instrument from which the opcode was called.

Use the *setksmps* opcode to set the local *ksmps* value.

The *xin* and *xout* opcodes copy variables to and from the opcode definition, allowing communication with the calling instrument.

The types of input and output variables are defined by the parameters *intypes* and *outtypes*.



### Tip

You can create UDOs which take no inputs or outputs by using 0 instead of a string.



### Notes

- *xin* and *xout* should be called only once, and *xin* should precede *xout*, otherwise an init error and deactivation of the current instrument may occur.
- These opcodes actually run only at i-time. Performance time copying is done by the user opcode call. This means that skipping *xin* or *xout* with *kgoto* has no effect, while skipping with *igoto* affects both init and performance time operation.

## Syntax

`opcode name, outtypes, intypes`

## Initialization

*name* -- name of the opcode. It may consist of any combination of letters, digits, and underscore but should not begin with a digit. If an opcode with the specified name already exists, it is redefined (a warning is printed in such cases). Some reserved words (like *instr* and *endin*) cannot be redefined.

*intypes* -- list of input types, any combination of the characters: a, k, O, P, V, K, i, o, p, and j. A single 0 character can be used if there are no input arguments. Double quotes and delimiter characters (e.g. comma) are *not* needed.

The meaning of the various *intypes* is shown in the following table:

Type	Description	Variable Types Allowed	Updated At
a	a-rate variable	a-rate	a-rate
i	i-rate variable	i-rate	i-time
j	optional i-time, defaults to -1	i-rate, constant	i-time
k	k-rate variable	k- and i-rate, constant	k-rate
O	optional k-rate variable, defaults to 0	k- and i-rate, constant	k-rate
P	optional k-rate variable, defaults to 1	k- and i-rate, constant	k-rate
V	optional k-rate variable, defaults to 0.5	k- and i-rate, constant	k-rate
K	k-rate with initialization	k- and i-rate, constant	i-time and k-rate
o	optional i-time, defaults to 0	i-rate, constant	i-time
p	optional i-time, defaults to 1	i-rate, constant	i-time
S	string variable	i-rate string	i-time

The maximum allowed number of input arguments is 256.

*outtypes* -- list of output types. The format is the same as in the case of *intypes*.

Here are the available *outtypes*:

Type	Description	Variable Types Allowed	Updated At
a	a-rate variable	a-rate	a-rate
i	i-rate variable	i-rate	i-time
k	k-rate variable	k-rate	k-rate
K	k-rate with initialization	k-rate	i-time and k-rate

The maximum allowed number of output arguments is 256.

*iksmips* (optional, default=0) -- sets the local *ksmps* value. Must be a positive integer, and also the *ksmps* of the calling instrument or opcode must be an integer multiple of this value. For example, if *ksmps* is 10 in the instrument from which the opcode was called, the allowed values for *iksmips* are 1, 2, 5, and 10.

If *iksmips* is set to zero, the *ksmps* of the caller instrument or opcode is used (this is the default behavior).



## Note

The local *ksmps* is implemented by splitting up a control period into smaller sub-k-periods and temporarily modifying internal Csound global variables. This also requires converting the rate of k-rate input and output arguments (input variables receive the same value in all sub-k-periods, while outputs are written only in the last one).



## Warning about local *ksmps*

When the local *ksmps* is not the same as the orchestra level *ksmps* value (as specified in the orchestra header), global a-rate operations must not be used in the user-defined opcode block.

These include:

- any access to “ga” variables
- a-rate zak opcodes (*zar*, *zaw*, etc.)
- *tablera* and *tablewa* (these two opcodes may in fact work, but caution is needed)
- The *in* and *out* opcode family (these read from, and write to global a-rate buffers)

In general, the local *ksmps* should be used with care as it is an experimental feature, although it works correctly in most cases.

The *setksmps* statement can be used to set the local *ksmps* value of the user-defined opcode block. It has one i-time parameter specifying the new *ksmps* value (which is left unchanged if zero is used, see also the notes about *iksmips* above). *setksmps* should be used before any other opcodes (but allowed after *xin*), otherwise unpredictable results may occur.

The input parameters can be read with *xin*, and the output is written by *xout* opcode. Only one instance of these units should be used, as *xout* overwrites and does not accumulate the output. The number and type of arguments for *xin* and *xout* must be the same as in the declaration of the user-defined opcode block (see tables above).

The input and output arguments must agree with the definition both in number (except if the optional i-time input is used) and type. An optional i-time input parameter (*iksmips*) is automatically added to the *intypes* list, and (similarly to *setksmps*) sets the local *ksmps* value.

## Performance

The syntax of a user-defined opcode block is as follows:

```
opcode name, outtypes, intypes
xinarg1 [, xinarg2] [, xinarg3] ... [xinargN] xin
[setksmps iksmips]
... the rest of the instrument's code.
xout xoutarg1 [, xoutarg2] [, xoutarg3] ... [xoutargN]
```

endop

The new opcode can then be used with the usual syntax:

```
[xoutarg1] [, xoutarg2] ... [xoutargN] name [xinarg1] [, xinarg2] ... [xinargN] [, ksmps]
```



## Note

The opcode call is always executed both at initialization and performance time, even if there are no a- or k-rate arguments. If there are many user opcode calls that are known to have no effect at performance time in an instrument, then it may save some CPU time to jump over groups of such opcodes with *kgoto*.

## Examples

Here is an example of a user-defined opcode. It uses the file *opcode.csd* [examples/opcode\_example.csd].

### Example 616. Example of a user-defined opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o opcode_example.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

/* example opcode 1: simple oscillator */

opcode Oscillator, a, kk

kamp, kcps      xin          ; read input parameters
a1      vco2 kamp, kcps      ; sawtooth oscillator
xout a1          ; write output

endop

/* example opcode 2: lowpass filter with local ksmps */

opcode Lowpass, a, akk

setksmps 1          ; need sr=kr
ain, ka1, ka2      xin          ; read input parameters
aout      init 0          ; initialize output
aout      = ain*ka1 + aout*ka2 ; simple tone-like filter
xout aout          ; write output

endop

/* example opcode 3: recursive call */
```

```

opcode RecursiveLowpass, a, akkpp

ain, ka1, ka2, idep, icnt      xin      ; read input parameters
      if (icnt >= idep) goto skip1    ; check if max depth reached
ain      RecursiveLowpass ain, ka1, ka2, idep, icnt + 1
skip1:
aout     Lowpass ain, ka1, ka2      ; call filter
      xout aout                    ; write output

      endop

/* example opcode 4: de-click envelope */

opcode DeClick, a, a

ain      xin
aenv     linseg 0, 0.02, 1, p3 - 0.05, 1, 0.02, 0, 0.01, 0
      xout ain * aenv              ; apply envelope and write output

      endop

/* instr 1 uses the example opcodes */

instr 1

kamp     = .7                      ; amplitude
kcps     expon 50, p3, 500        ; pitch
a1       Oscillator kamp, kcps      ; call oscillator
kflt     linseg 0.4, 1.5, 0.4, 1, 0.8, 1.5, 0.8 ; filter envelope
a1       RecursiveLowpass a1, kflt, 1 - kflt, 10 ; 10th order lowpass
a1       DeClick a1
      outs a1, a1

      endin

</CsInstruments>
<CsScore>

i 1 0 4
e5 ;extra second before quitting

</CsScore>
</CsoundSynthesizer>

```

## See Also

*endop, setksmps, xin, xout*

## Credits

Author: Istvan Varga, 2002; based on code by Matt J. Ingalls

New in version 4.22

# oscbnk

oscbnk — Mixes the output of any number of oscillators.

## Description

This unit generator mixes the output of any number of oscillators. The frequency, phase, and amplitude of each oscillator can be modulated by two LFOs (all oscillators have a separate set of LFOs, with different phase and frequency); additionally, the output of each oscillator can be filtered through an optional parametric equalizer (also controlled by the LFOs). This opcode is most useful for rendering ensemble (strings, choir, etc.) instruments.

Although the LFOs run at k-rate, amplitude, phase and filter modulation are interpolated internally, so it is possible (and recommended in most cases) to use this unit at low (~1000 Hz) control rates without audible quality degradation.

The start phase and frequency of all oscillators and LFOs can be set by a built-in seedable 31-bit random number generator, or specified manually in a function table (GEN2).

## Syntax

```
ares oscbnk kcps, kamd, kfmd, kpmf, iovrlap, iseed, kllminf, kllmaxf, \  
k12minf, k12maxf, ilfomode, keqminf, keqmaxf, keqmin1, keqmax1, \  
keqminq, keqmaxq, ieqmode, kfn [, il1fn] [, il2fn] [, ieqffn] \  
[, ieq1fn] [, ieqqfn] [, itabl] [, ioutfn]
```

## Initialization

*iovrlap* -- Number of oscillator units.

*iseed* -- Seed value for random number generator (positive integer in the range 1 to 2147483646 ( $2^{31} - 2$ )). *iseed*  $\leq 0$  seeds from the current time.

*ieqmode* -- Parametric equalizer mode

- -1: disable EQ (faster)
- 0: peak
- 1: low shelf
- 2: high shelf
- 3: peak (filter interpolation disabled)
- 4: low shelf (interpolation disabled)
- 5: high shelf (interpolation disabled)

The non-interpolated modes are faster, and in some cases (e.g. high shelf filter at low cutoff frequencies) also more stable; however, interpolation is useful for avoiding “zipper noise” at low control rates.

*ilfomode* -- LFO modulation mode, sum of:

- 128: LFO1 to frequency
- 64: LFO1 to amplitude

- 32: LFO1 to phase
- 16: LFO1 to EQ
- 8: LFO2 to frequency
- 4: LFO2 to amplitude
- 2: LFO2 to phase
- 1: LFO2 to EQ

If an LFO does not modulate anything, it is not calculated, and the *ftable* number (*il1fn* or *il2fn*) can be omitted.

*il1fn* (optional: default=0) -- LFO1 function table number. The waveform in this table has to be normalized (absolute value  $\leq 1$ ), and is read with linear interpolation.

*il2fn* (optional: default=0) -- LFO2 function table number. The waveform in this table has to be normalized, and is read with linear interpolation.

*ieqffn*, *ieqlfn*, *ieqqfn* (optional: default=0) -- Lookup tables for EQ frequency, level, and Q (optional if EQ is disabled). Table read position is 0 if the modulator signal is less than, or equal to -1, (table length / 2) if the modulator signal is zero, and the guard point if the modulator signal is greater than, or equal to 1. These tables have to be normalized to the range 0 - 1, and have an extended guard point (table length = power of two + 1). All tables are read with linear interpolation.

*itabl* (optional: default=0) -- Function table storing phase and frequency values for all oscillators (optional). The values in this table are in the following order (5 for each oscillator unit):

oscillator phase, lfo1 phase, lfo1 frequency, lfo2 phase, lfo2 frequency, ...

All values are in the range 0 to 1; if the specified number is greater than 1, it is wrapped (phase) or limited (frequency) to the allowed range. A negative value (or end of table) will use the output of the random number generator. The random seed is always updated (even if no random number was used), so switching one value between random and fixed will not change others.

*ioutfn* (optional: default=0) -- Function table to write phase and frequency values (optional). The format is the same as in the case of *itabl*. This table is useful when experimenting with random numbers to record the best values.

The two optional tables (*itabl* and *ioutfn*) are accessed only at i-time. This is useful to know, as the tables can be safely overwritten after opcode initialization, which allows precalculating parameters at i-time and storing in a temporary table before *oscbnk* initialization.

## Performance

*ares* -- Output signal.

*kcps* -- Oscillator frequency in Hz.

*kamd* -- AM depth (0 - 1).

(AM output) = (AM input) \* ((1 - (AM depth)) + (AM depth) \* (modulator))

If *ilfomode* isn't set to modulate the amplitude, then (AM output) = (AM input) regardless of the value of *kamd*. That means that *kamd* will have no effect.

Note: Amplitude modulation is applied before the parametric equalizer.

*kfmd* -- FM depth (in Hz).

*kpm* -- Phase modulation depth.

*kl1minf*, *kl1maxf* -- LFO1 minimum and maximum frequency in Hz.

*kl2minf*, *kl2maxf* -- LFO2 minimum and maximum frequency in Hz. (Note: oscillator and LFO frequencies are allowed to be zero or negative.)

*keqminf*, *keqmaxf* -- Parametric equalizer minimum and maximum frequency in Hz.

*keqminl*, *keqmaxl* -- Parametric equalizer minimum and maximum level.

*keqminq*, *keqmaxq* -- Parametric equalizer minimum and maximum Q.

*kfn* -- Oscillator waveform table. Table number can be changed at k-rate (this is useful to select from a set of band-limited tables generated by GEN30, to avoid aliasing). The table is read with linear interpolation.



## Note

*oscbnk* uses the same random number generator as *rnd31*. So reading *its documentation* is also recommended.

## Examples

Here is an example of *oscbnk* opcode. It uses the file *oscbnk.csd* [examples/oscbnk.csd].

### Example 617. Example of the *oscbnk* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o oscbnk.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

/* Written by Istvan Varga */
sr = 48000
kr = 750
ksmps = 64
nchnls = 2

ga01 init 0
ga02 init 0

/* sawtooth wave */
i_ ftgen 1, 0, 16384, 7, 1, 16384, -1
/* FM waveform */
i_ ftgen 3, 0, 4096, 7, 0, 512, 0.25, 512, 1, 512, 0.25, 512, \
    0, 512, -0.25, 512, -1, 512, -0.25, 512, 0
/* AM waveform */
i_ ftgen 4, 0, 4096, 5, 1, 4096, 0.01
/* FM to EQ */
i_ ftgen 5, 0, 1024, 5, 1, 512, 32, 512, 1
```

```

/* sine wave */
i_ ftgen 6, 0, 1024, 10, 1
/* room parameters */
i_ ftgen 7, 0, 64, -2, 4, 50, -1, -1, -1, 11, \
    1, 26.833, 0.05, 0.85, 10000, 0.8, 0.5, 2, \
    1, 1.753, 0.05, 0.85, 5000, 0.8, 0.5, 2, \
    1, 39.451, 0.05, 0.85, 7000, 0.8, 0.5, 2, \
    1, 33.503, 0.05, 0.85, 7000, 0.8, 0.5, 2, \
    1, 36.151, 0.05, 0.85, 7000, 0.8, 0.5, 2, \
    1, 29.633, 0.05, 0.85, 7000, 0.8, 0.5, 2

/* generate bandlimited sawtooth waves */

i0 = 0
loop1:
imaxh = sr / (2 * 440.0 * exp(log(2.0) * (i0 - 69) / 12))
i_ ftgen i0 + 256, 0, 4096, -30, 1, 1, imaxh
i0 = i0 + 1
    if (i0 < 127.5) igoto loop1

instr 1

p3 = p3 + 0.4

; note frequency
kcps = 440.0 * exp(log(2.0) * (p4 - 69) / 12)
; lowpass max. frequency
klpmaxf limit 64 * kcps, 1000.0, 12000.0
; FM depth in Hz
kfmd1 = 0.02 * kcps
; AM frequency
kamfr = kcps * 0.02
kamfr2 = kcps * 0.1
; table number
kfnum = (256 + 69 + 0.5 + 12 * log(kcps / 440.0) / log(2.0))
; amp. envelope
aenv linseg 0, 0.1, 1.0, p3 - 0.5, 1.0, 0.1, 0.5, 0.2, 0, 1.0, 0

/* oscillator / left */

a1 oscbnk kcps, 0.0, kfmd1, 0.0, 40, 200, 0.1, 0.2, 0, 0, 144, \
    0.0, klpmaxf, 0.0, 0.0, 1.5, 1.5, 2, \
    kfnum, 3, 0, 5, 5, 5
a2 oscbnk kcps, 1.0, kfmd1, 0.0, 40, 201, 0.1, 0.2, kamfr, kamfr2, 148, \
    0, 0, 0, 0, 0, 0, -1, \
    kfnum, 3, 4
a2 pareq a2, kcps * 8, 0.0, 0.7071, 2
a0 = a1 + a2 * 0.12
/* delay */
adel = 0.001
a01 vdelayx a0, adel, 0.01, 16
a_ oscili 1.0, 0.25, 6, 0.0
adel = adel + 1.0 / (exp(log(2.0) * a_) * 8000)
a02 vdelayx a0, adel, 0.01, 16
a0 = a01 + a02

ga01 = ga01 + a0 * aenv * 2500

/* oscillator / right */

; lowpass max. frequency

a1 oscbnk kcps, 0.0, kfmd1, 0.0, 40, 202, 0.1, 0.2, 0, 0, 144, \
    0.0, klpmaxf, 0.0, 0.0, 1.0, 1.0, 2, \
    kfnum, 3, 0, 5, 5, 5
a2 oscbnk kcps, 1.0, kfmd1, 0.0, 40, 203, 0.1, 0.2, kamfr, kamfr2, 148, \
    0, 0, 0, 0, 0, 0, -1, \

```

```

    kfnum, 3, 4
a2 pareq a2, kcps * 8, 0.0, 0.7071, 2
a0 = a1 + a2 * 0.12
/* delay */
adel = 0.001
a01 vdelayx a0, adel, 0.01, 16
a_ oscili 1.0, 0.25, 6, 0.25
adel = adel + 1.0 / (exp(log(2.0) * a_) * 8000)
a02 vdelayx a0, adel, 0.01, 16
a0 = a01 + a02

ga02 = ga02 + a0 * aenv * 2500

    endin

/* output / left */

    instr 81

i1 = 0.000001
aLl, aLh, aRl, aRh spat3di ga01 + i1*i1*i1*i1, -8.0, 4.0, 0.0, 0.3, 7, 4
ga01 = 0
aLl butterlp aLl, 800.0
aRl butterlp aRl, 800.0

outs aLl + aLh, aRl + aRh

    endin

/* output / right */

    instr 82

i1 = 0.000001
aLl, aLh, aRl, aRh spat3di ga02 + i1*i1*i1*i1, 8.0, 4.0, 0.0, 0.3, 7, 4
ga02 = 0
aLl butterlp aLl, 800.0
aRl butterlp aRl, 800.0

outs aLl + aLh, aRl + aRh

    endin

</CsInstruments>
<CsScore>

/* Written by Istvan Varga */
t 0 60

i 1 0 4 41
i 1 0 4 60
i 1 0 4 65
i 1 0 4 69

i 81 0 5.5
i 82 0 5.5
e

</CsScore>
</CsoundSynthesizer>

```

## Credits

Author: Istvan Varga  
2001

New in version 4.15

Updated April 2002 by Istvan Varga

# oscil1

oscil1 — Accesses table values by incremental sampling.

## Description

Accesses table values by incremental sampling.

## Syntax

```
kres oscil1 idel, kamp, idur [, ifn]
```

## Initialization

*idel* -- delay in seconds before *oscil1* incremental sampling begins.

*idur* -- duration in seconds to sample through the *oscil1* table just once. A negative value will make the table be read from the end to the beginning.

*ifn* -- (optional) function table number. *tablei*, *oscilli* require the extended guard point. The number defaults to -1 which indicates a sinewave.

## Performance

*kamp* -- amplitude factor.

*oscil1* accesses values by sampling once through the function table at a rate determined by *idur*. For the first *idel* seconds, the point of scan will reside at the first location of the table; it will then begin moving through the table at a constant rate, reaching the end in another *idur* seconds; from that time on (i.e. after *idel* + *idur* seconds) it will remain pointing at the last location. Each value obtained from sampling is then multiplied by an amplitude factor *kamp* before being written into the result.

## Examples

Here is an example of the *oscil1* opcode. It uses the file *oscil1.csd* [examples/oscil1.csd].

### Example 618. Example of the *oscil1* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o oscilli.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
```

```
ksmps = 32
nchnls = 2
0dbfs = 1

instr      1

ipanfn = p4
asig      vco2 .3, 220
kpan      oscilli 0, 1, p3, ipanfn ;create panning &
kleft     = sqrt(kpan) ;start right away
kright    = sqrt(1-kpan)
outs      kleft*asig, kright*asig

endin
</CsInstruments>
<CsScore>
f 1 0 3 -7 .5 3 .5 ;remain in center (.5 CONSTANT)
f 2 0 129 7 1 129 0 ;left-->right
f 3 0 129 7 .5 32 1 64 0 33 .5 ;center-->left-->right-->center

i 1 0 2 1 ;use table 1
i 1 3 2 2 ;use table 2
i 1 6 2 3 ;use table 3

e
</CsScore>
</CsSoundSynthesizer>
```

The example will produce the following output:

```
i1      0.50000
i1      0.20000
i1      0.80000
i1      0.10000
i1      0.90000
i1      0.00000
i1      1.00000
i1      0.50000
```

## See Also

*table, tablei, table3, oscilli, osciln*

# oscil1i

oscil1i — Accesses table values by incremental sampling with linear interpolation.

## Description

Accesses table values by incremental sampling with linear interpolation.

## Syntax

```
kres oscil1i idel, kamp, idur [, ifn]
```

## Initialization

*idel* -- delay in seconds before *oscil1i* incremental sampling begins.

*idur* -- duration in seconds to sample through the *oscil1i* table just once. A negative value will make the table be read from the end to the beginning.

*ifn* -- (optional) function table number. *oscil1i* requires the extended guard point. The default value is -1 indicating a sine wave.

## Performance

*kamp* -- amplitude factor

*oscil1i* is an interpolating unit in which the fractional part of index is used to interpolate between adjacent table entries. The smoothness gained by interpolation is at some small cost in execution time (see also *oscili*, etc.), but the interpolating and non-interpolating units are otherwise interchangeable.

## Examples

Here is an example of the *oscil1i* opcode. It uses the file *oscil1i.csd* [examples/oscil1i.csd].

### Example 619. Example of the *oscil1i* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o oscil1i.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
```

```
instr      1

  ipanfn = p4
  asig   vco2 .3, 220
  kpan   oscilli 0, 1, p3, ipanfn ;create panning &
  kleft  = sqrt(kpan) ;start right away
  kright = sqrt(1-kpan)
         outs kleft*asig, kright*asig

  endin
</CsInstruments>
<CsScore>
f 1 0 3 -7 .5 3 .5 ;remain in center (.5 CONSTANT)
f 2 0 129 7 1 129 0 ;left-->right
f 3 0 129 7 .5 32 1 64 0 33 .5 ;center-->left-->right-->center

i 1 0 2 1 ;use table 1
i 1 3 2 2 ;use table 2
i 1 6 2 3 ;use table 3

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*table, tablei, table3, oscill, osciln*

# oscil3

oscil3 — A simple oscillator with cubic interpolation.

## Description

*oscil3* reads table *ifn* sequentially and repeatedly at a frequency *xcps*. The amplitude is scaled by *xamp*. Cubic interpolation is applied for table look up from internal phase values.

## Syntax

```
ares oscil3 xamp, xcps [, ifn, iphs]
kres oscil3 kamp, kcps [, ifn, iphs]
```

## Initialization

*ifn* (optional) -- function table number. Requires a wrap-around guard point. Defaults to -1 indicating a sinewave.

*iphs* (optional) -- initial phase of sampling, expressed as a fraction of a cycle (0 to 1). A negative value will cause phase initialization to be skipped. The default value is 0.

## Performance

*kamp*, *xamp* -- amplitude

*kcps*, *xcps* -- frequency in cycles per second.

*oscil3* is identical to *oscili*, except that it uses cubic interpolation.

Table *ifn* is incrementally sampled modulo the table length and the value obtained is multiplied by *amp*.

If you need to change the oscillator table with a k-rate signal, you can use *oscilikt*.

## Examples

Here is an example of the *oscil3* opcode. It uses the file *oscil3.csd* [examples/oscil3.csd].

### Example 620. Example of the *oscil3* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o oscil3.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kamp = .6
kcps = 440
ifn = p4

asig oscil kamp, kcps, ifn
outs asig,asig

endin

instr 2

kamp = .6
kcps = 440
ifn = p4

asig oscil3 kamp, kcps, ifn
outs asig,asig

endin
</CsInstruments>
<CsScore>
f1 0 128 10 1 ; Sine with a small amount of data
f2 0 128 10 1 0.5 0.3 0.25 0.2 0.167 0.14 0.125 .111 ; Sawtooth with a small amount of data
f3 0 128 10 1 0 0.3 0 0.2 0 0.14 0 .111 ; Square with a small amount of data
f4 0 128 10 1 1 1 1 0.7 0.5 0.3 0.1 ; Pulse with a small amount of data

i 1 0 2 1
i 2 3 2 1
i 1 6 2 2
i 2 9 2 2
i 1 12 2 3
i 2 15 2 3
i 1 18 2 4
i 2 21 2 4

e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*oscil, oscili, oscilikt*

## Credits

Author: John ffitth

New in Csound version 3.50

# oscil

oscil — A simple oscillator.

## Description

*oscil* reads table *ifn* sequentially and repeatedly at a frequency *xcps*. The amplitude is scaled by *xamp*.

## Syntax

```
ares oscil xamp, xcps [ , ifn, iphs]
```

```
kres oscil kamp, kcps [ , ifn, iphs]
```

## Initialization

*ifn* -- (optional) function table number. Requires a wrap-around guard point. The table number defaults to -1 which indicates a sinewave.

*iphs* (optional, default=0) -- initial phase of sampling, expressed as a fraction of a cycle (0 to 1). A negative value will cause phase initialization to be skipped. The default value is 0.

## Performance

*kamp*, *xamp* -- amplitude

*kcps*, *xcps* -- frequency in cycles per second.

The *oscil* opcode generates periodic control (or audio) signals consisting of the value of *kamp* (*xamp*) times the value returned from control rate (audio rate) sampling of a stored function table. The internal phase is simultaneously advanced in accordance with the *kcps* or *xcps* input value.

Table *ifn* is incrementally sampled modulo the table length and the value obtained is multiplied by *amp*.

If you need to change the oscillator table with a k-rate signal, you can use *oscilikt*.

## Examples

Here is an example of the *oscil* opcode. It uses the file *oscil.csd* [examples/oscil.csd].

### Example 621. Example of the *oscil* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o oscil.wav -W ;; for file output any platform
```

```
</CsOptions>
<CsInstruments>

  sr = 44100
  ksmps = 32
  nchnls = 2
  0dbfs = 1

  instr 1

  kamp = .6
  kcps = 440
  ifn = p4

  asig oscil kamp, kcps, ifn
      outs asig,asig

  endin
</CsInstruments>
<CsScore>
f1 0 16384 10 1 ; Sine
f2 0 16384 10 1 0.5 0.3 0.25 0.2 0.167 0.14 0.125 .111 ; Sawtooth
f3 0 16384 10 1 0 0.3 0 0.2 0 0.14 0 .111 ; Square
f4 0 16384 10 1 1 1 1 0.7 0.5 0.3 0.1 ; Pulse

i 1 0 2 1
i 1 3 2 2
i 1 6 2 3
i 1 9 2 4

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*oscili, oscilikt, oscil3, poscil, poscil3*

# oscili

oscili — A simple oscillator with linear interpolation.

## Description

*oscili* reads table *ifn* sequentially and repeatedly at a frequency *xcps*. The amplitude is scaled by *xamp*. Linear interpolation is applied for table look up from internal phase values.

## Syntax

```
ares oscili xamp, xcps[, ifn, iphs]
kres oscili kamp, kcps[, ifn, iphs]
```

## Initialization

*ifn*(optional) -- function table number. Requires a wrap-around guard point. The table number defaults to -1 which indicates a sine wave.

*iphs* (optional) -- initial phase of sampling, expressed as a fraction of a cycle (0 to 1). A negative value will cause phase initialization to be skipped. The default value is 0.

## Performance

*kamp*, *xamp* -- amplitude

*kcps*, *xcps* -- frequency in cycles per second.

*oscili* differs from *oscil* in that the standard procedure of using a truncated phase as a sampling index is here replaced by a process that interpolates between two successive lookups. Interpolating generators will produce a noticeably cleaner output signal, but they may take as much as twice as long to run. Adequate accuracy can also be gained without the time cost of interpolation by using large stored function tables of 2K, 4K or 8K points if the space is available.

Table *ifn* is incrementally sampled modulo the table length and the value obtained is multiplied by *amp*.

If you need to change the oscillator table with a k-rate signal, you can use *oscilikt*.

## Examples

Here is an example of the *oscili* opcode. It uses the file *oscili.csd* [examples/oscili.csd].

### Example 622. Example of the *oscili* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
```

```

;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o oscili.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kamp = .6
kcps = 440
ifn = p4

asig oscil kamp, kcps, ifn
outs asig,asig

endin

instr 2

kamp = .6
kcps = 440
ifn = p4

asig oscili kamp, kcps, ifn
outs asig,asig

endin
</CsInstruments>
<CsScore>
f1 0 128 10 1 ; Sine with a small amount of data
f2 0 128 10 1 0.5 0.3 0.25 0.2 0.167 0.14 0.125 .111 ; Sawtooth with a small amount of data
f3 0 128 10 1 0 0.3 0 0.2 0 0.14 0 .111 ; Square with a small amount of data
f4 0 128 10 1 1 1 1 0.7 0.5 0.3 0.1 ; Pulse with a small amount of data

i 1 0 2 1
i 2 3 2 1
i 1 6 2 2
i 2 9 2 2
i 1 12 2 3
i 2 15 2 3
i 1 18 2 4
i 2 21 2 4

e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*oscil*, *oscil3*

## Credits

# oscilikt

oscilikt — A linearly interpolated oscillator that allows changing the table number at k-rate.

## Description

*oscilikt* is very similar to *oscili*, but allows changing the table number at k-rate. It is slightly slower than *oscili* (especially with high control rate), although also more accurate as it uses a 31-bit phase accumulator, as opposed to the 24-bit one used by *oscili*.

## Syntax

```
ares oscilikt xamp, xcps, kfn [, iphs] [, istor]
kres oscilikt kamp, kcps, kfn [, iphs] [, istor]
```

## Initialization

*iphs* (optional, defaults to 0) -- initial phase in the range 0 to 1. Other values are wrapped to the allowed range.

*istor* (optional, defaults to 0) -- skip initialization.

## Performance

*kamp*, *xamp* -- amplitude.

*kcps*, *xcps* -- frequency in Hz. Zero and negative values are allowed. However, the absolute value must be less than *sr* (and recommended to be less than  $sr/2$ ).

*kfn* -- function table number. Can be varied at control rate (useful to “morph” waveforms, or select from a set of band-limited tables generated by *GEN30*).

## Examples

Here is an example of the *oscilikt* opcode. It uses the file *oscilikt.csd* [examples/oscilikt.csd].

### Example 623. Example of the *oscilikt* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsoundOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in   No messages
-odac          -iadc      -d          ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o oscilikt.wav -W ;; for file output any platform
</CsoundOptions>
<CsoundInstruments>

; Initialize the global variables.
```

```
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Generate a uni-polar (0-1) square wave.
  kamp1 init 1
  kcps1 init 2
  itype = 3
  ksquare lfo kamp1, kcps1, itype

  ; Use the square wave to switch between Tables #1 and #2.
  kamp2 init 20000
  kcps2 init 220
  kfn = ksquare + 1

  a1 oscilikt kamp2, kcps2, kfn
  out a1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine waveform.
f 1 0 4096 10 0 1
; Table #2: a sawtooth wave
f 2 0 3 -2 1 0 -1

; Play Instrument #1 for two seconds.
i 1 0 2

</CsScore>
</CsSoundSynthesizer>
```

## See Also

*oscilikt* and *oscilikts*.

## Credits

Author: Istvan Varga

Example written by Kevin Conder.

New in version 4.22

# osciliktp

osciliktp — A linearly interpolated oscillator that allows allows phase modulation.

## Description

*osciliktp* allows phase modulation (which is actually implemented as k-rate frequency modulation, by differentiating phase input). The disadvantage is that there is no amplitude control, and frequency can be varied only at the control-rate. This opcode can be faster or slower than *oscilikt*, depending on the control-rate.

## Syntax

```
ares osciliktp kcps, kfn, kphs [, istor]
```

## Initialization

*istor* (optional, defaults to 0) -- Skips initialization.

## Performance

*ares* -- audio-rate ouput signal.

*kcps* -- frequency in Hz. Zero and negative values are allowed. However, the absolute value must be less than *sr* (and recommended to be less than *sr/2*).

*kfn* -- function table number. Can be varied at control rate (useful to “morph” waveforms, or select from a set of band-limited tables generated by *GEN30*).

*kphs* -- phase (k-rate), the expected range is 0 to 1. The absolute value of the difference of the current and previous value of *kphs* must be less than *ksmps*.

## Examples

Here is an example of the *osciliktp* opcode. It uses the file *osciliktp.csd* [examples/osciliktp.csd].

### Example 624. Example of the *osciliktp* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc      -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o osciliktp.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
```

```
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1: osciliktp example
instr 1
  kphs line 0, p3, 4

  a1x osciliktp 220.5, 1, 0
  a1y osciliktp 220.5, 1, -kphs
  a1 = a1x - a1y

  out a1 * 14000
endin

</CsInstruments>
<CsScore>

; Table #1: Sawtooth wave
f 1 0 3 -2 1 0 -1

; Play Instrument #1 for four seconds.
i 1 0 4
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*oscilikt* and *oscilikts*.

## Credits

Author: Istvan Varga

New in version 4.22

# oscilikts

oscilikts — A linearly interpolated oscillator with sync status that allows changing the table number at k-rate.

## Description

*oscilikts* is the same as *oscilikt*. Except it has a sync input that can be used to re-initialize the oscillator to a k-rate phase value. It is slower than *oscilikt* and *osciliktp*.

## Syntax

```
ares oscilikts xamp, xcps, kfn, async, kphs [, istor]
```

## Initialization

*istor* (optional, defaults to 0) -- skip initialization.

## Performance

*xamp* -- amplitude.

*xcps* -- frequency in Hz. Zero and negative values are allowed. However, the absolute value must be less than *sr* (and recommended to be less than  $sr/2$ ).

*kfn* -- function table number. Can be varied at control rate (useful to “morph” waveforms, or select from a set of band-limited tables generated by *GEN30*).

*async* -- any positive value resets the phase of *oscilikts* to *kphs*. Zero or negative values have no effect.

*kphs* -- sets the phase, initially and when it is re-initialized with *async*.

## Examples

Here is an example of the *oscilikts* opcode. It uses the file *oscilikts.csd* [examples/oscilikts.csd].

### Example 625. Example of the *oscilikts* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o oscilikts.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
```

```
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1: oscilikts example.
instr 1
; Frequency envelope.
kfrq expon 400, p3, 1200
; Phase.
kphs line 0.1, p3, 0.9

; Sync 1
atmp1 phasor 100
; Sync 2
atmp2 phasor 150
async diff 1 - (atmp1 + atmp2)

a1 oscilikts 14000, kfrq, 1, async, 0
a2 oscilikts 14000, kfrq, 1, async, -kphs

out a1 - a2
endin

</CsInstruments>
<CsScore>

; Table #1: Sawtooth wave
f 1 0 3 -2 1 0 -1

; Play Instrument #1 for four seconds.
i 1 0 4
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*oscilikt* and *osciliktp*.

## Credits

Author: Istvan Varga

New in version 4.22

# osciln

osciln — Accesses table values at a user-defined frequency.

## Description

Accesses table values at a user-defined frequency. This opcode can also be written as *oscilx*.

## Syntax

```
ares osciln kamp, ifrq, ifn, itimes
```

## Initialization

*ifrq, itimes* -- rate and number of times through the stored table.

*ifn* -- function table number.

## Performance

*kamp* -- amplitude factor

*osciln* will sample several times through the stored table at a rate of *ifrq* times per second, after which it will output zeros. Generates audio signals only, with output values scaled by *kamp*.

## Examples

Here is an example of the *osciln* opcode. It uses the file *osciln.csd* [examples/osciln.csd].

### Example 626. Example of the *osciln* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o osciln.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gione ftgen 1, 0, 1024, 7, 0, 1, 1, 1024, 0
gitwo ftgen 2, 0, 1024, 7, 0, 512, 1, 512, 0

instr 1 ;very simple waveguide system
```

```
ifn      = p4
ipitch   = p5
itimes   = p6
iperiod  = 1000/ipitch

afeed    init    0
aimpl    osciln 1, ipitch, ifn, itimes ;use as excitation signal
arefl    tone   aimpl + afeed, 4000
aout     atone  arefl, 5000
afeed    vdelay arefl, iperiod, 10
          outs  aout*3, aout*3

endin
</CsInstruments>
<CsScore>

i 1 0 4 1 110 1 ;use different tables,
i 1 5 4 2 110 1 ;& different pitch
i 1 10 4 1 110 10 ;& different number of times the table is read
i 1 15 4 2 110 10
i 1 20 6 1 880 1
i 1 25 3 2 880 1
i 1 30 3 1 880 10
i 1 35 3 2 880 10

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*table, tablei, table3, oscill, oscilli*

# oscils

oscils — A simple, fast sine oscillator

## Description

Simple, fast sine oscillator, that uses only one multiply, and two add operations to generate one sample of output, and does not require a function table.

## Syntax

```
ares oscils iamp, icps, iphs [, iflg]
```

## Initialization

*iamp* -- output amplitude.

*icps* -- frequency in Hz (may be zero or negative, however the absolute value must be less than  $sr/2$ ).

*iphs* -- start phase between 0 and 1.

*iflg* -- sum of the following values:

- 2: use double precision even if Csound was compiled to use floats. This improves quality (especially in the case of long performance time), but may be up to twice as slow.
- 1: skip initialization.

## Performance

*ares* -- audio output

## Examples

Here is an example of the oscils opcode. It uses the file *oscils.csd* [examples/oscils.csd].

### Example 627. Example of the oscils opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o oscils.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
```

```
nchnls = 2
odbfs = 1

instr 1

iflg = p4
asig oscils .7, 220, 0, iflg
outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 2 0
i 1 3 2 2 ;double precision
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Istvan Varga  
January 2002

New in version 4.18

## **oscilx**

oscilx — Same as the osciln opcode.

### **Description**

Same as the *osciln* opcode.

# OSCinit

OSCinit — Start a listening process for OSC messages to a particular port.

## Description

Starts a listening process, which can be used by OSClisten.

## Syntax

```
ihandle OSCinit iport
```

## Initialization

*ihandle* -- handle returned that can be passed to any number of OSClisten opcodes to receive messages on this port.

*iport* -- the port on which to listen.

## Performance

The listener runs in the background. See OSClisten for details.

## Example

The example shows a pair of floating point numbers being received on port 7770.

```
sr = 44100
ksmps = 100
nchnls = 2

gihandle OSCinit 7770

instr 1
  kf1 init 0
  kf2 init 0
nxtmsg:
  kk OSClisten gihandle, "/foo/bar", "ff", kf1, kf2
if (kk == 0) goto ex
  printk 0,kf1
  printk 0,kf2
  kgoto nxtmsg
ex:
  endin
```

## See Also

*OSClisten*, *OSCsend*

More information on this opcode: [http://www.youtube.com/watch?v=JX1C3TqP\\_9Y](http://www.youtube.com/watch?v=JX1C3TqP_9Y) , made by Andrés Cabrera

## Credits

Author: John ffitch  
2005

# OSClisten

OSClisten — Listen for OSC messages to a particular path.

## Description

On each k-cycle looks to see if an OSC message has been send to a given path of a given type.

## Syntax

```
kans OSClisten ihandle, idest, itype [, xdata1, xdata2, ...]
```

## Initialization

*ihandle* -- a handle returned by an earlier call to OSCinit, to associate OSClisten with a particular port number.

*idest* -- a string that is the destination address. This takes the form of a file name with directories. Csound uses this address to decide if messages are meant for csound.

*itype* -- a string that indicates the types of the optional arguments that are to be read. The string can contain the characters "acdfhisAG" which stand for audio, character, double, float, 64-bit integer, 32-bit integer, string, scalar array and f-table. All types other than 'asA' require a k-rate variable; 's' requires a string variable, 'a' an audio variable and 'A' an i- or k- rate array. For type 'G' a variable or constant can be used.

A handler is inserted into the listener (see OSCinit) to intercept messages of this pattern.

## Performance

*kans* -- set to 1 if a new message was received, or zero if not. If multiple messages are received in a single control period, the messages are buffered, and OSClisten can be called again until zero is returned.

If there was a message the *xdata* variables are set to the incoming values, as interpreted by the *itype* parameter. Note that although the *xdata* variables are on the right of an operation they are actually outputs, and so must be variables of type a, ga, k, gk, S, gS, k[] and gk[] and may need to be declared with init, or = in the case of string variables, before calling OSClisten.

## Example

The example shows a pair of floating point numbers being received on port 7770.

```
sr = 44100
ksmps = 100
nchnls = 2

gihandle OSCinit 7770

instr 1
  kf1 init 0
```

```

kf2 init 0
nextmsg:
kk OSClisten gihandle, "/foo/bar", "ff", kf1, kf2
if (kk == 0) goto ex
printk 0,kf1
printk 0,kf2
kgoto nextmsg
ex:
endin

```

Below are two .csd files which demonstrate the usage of the OSC opcodes. They use the files *OSCmidisend.csd* [examples/OSCmidisend.csd] and *OSCmidircv.csd* [examples/OSCmidircv.csd].

### Example 628. Example of the OSC opcodes.

The following two .csd files demonstrate the usage of the OSC opcodes in csound. The first file, *OSCmidisend.csd* [examples/OSCmidisend.csd], transforms received real-time MIDI messages into OSC data. The second file, *OSCmidircv.csd* [examples/OSCmidircv.csd], can take these OSC messages, and interpret them to generate sound from note messages, and store controller values. It will use controller number 7 to control volume. Note that these files are designed to be on the same machine, but if a different host address (in the IPADDRESS macro) is used, they can be separate machines on a network, or connected through the internet.

CSD file to send OSC messages:

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      -iadc      ;;RT audio I/O
</CsOptions>
<CsInstruments>

sr      = 44100
ksmps  = 128
nchnls = 1

; Example by David Akbari 2007
; Modified by Jonathan Murphy
; Use this file to generate OSC events for OSCmidircv.csd

#define IPADDRESS # "localhost" #
#define PORT      # 47120 #

turnon 1000

instr 1000

kst, kch, kd1, kd2  midiin

OSCsend  kst+kch+kd1+kd2, $IPADDRESS, $PORT, "/midi", "iiii", kst, kch, kd1, kd2

endin

</CsInstruments>
<CsScore>
f 0 3600 ;Dummy f-table
e
</CsScore>
</CsoundSynthesizer>

```

CSD file to receive OSC messages:

```

<CsoundSynthesizer>
; network_recv.csd
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      -iadc      ;;RT audio I/O
</CsOptions>
<CsInstruments>

  sr      = 44100
  ksmpls  = 128
  nchnls  = 1

; Example by Jonathan Murphy and Andres Cabrera 2007
; Use file OSCmidisend.csd to generate OSC events for this file

  Odbfs   = 1

  gilisten  OSCinit  47120

  gisin    ftgen     1, 0, 16384, 10, 1
  givel    ftgen     2, 0, 128, -2, 0
  gicc     ftgen     3, 0, 128, -7, 100, 128, 100 ;Default all controllers to 100

;Define scale tuning
  giji_12  ftgen     202, 0, 32, -2, 12, 2, 256, 60, 1, 16/15, 9/8, 6/5, 5/4, 4/3, 7/5, \
              3/2, 8/5, 5/3, 9/5, 15/8, 2

#define DEST #"/midi"#
; Use controller number 7 for volume
#define VOL #7#

turnon 1000

  instr 1000

  kst      init      0
  kch      init      0
  kd1      init      0
  kd2      init      0

next:

  kk       OSClisten gilisten, $DEST, "iiii", kst, kch, kd1, kd2

  if (kk == 0) goto done

  printks "kst = %i, kch = %i, kd1 = %i, kd2 = %i\\n", \
          0, kst, kch, kd1, kd2

  if (kst == 176) then
;Store controller information in a table
    tablew kd2, kd1, gicc
  endif

  if (kst == 144) then
;Process noteon and noteoff messages.
    kkey    = kd1
    kvel    = kd2
    kcps    cpstun  kvel, kkey, giji_12
    kamp    = kvel/127

  if (kvel == 0) then

```

```
        turnoff2 1001, 4, 1
elseif (kvel > 0) then
    event "i", 1001, 0, -1, kcps, kamp
endif
endif

kgoto next ;Process all events in queue

done:
    endin

instr 1001 ;Simple instrument

icps    init    p4
kvol    table   $VOL, gicc ;Read MIDI volume from controller table
kvol    =    kvol/127

aenv    linsegr  0, .003, p5, 0.03, p5 * 0.5, 0.3, 0
aosc    oscil   aenv, icps, gisin

    out    aosc * kvol
    endin

</CsInstruments>
<CsScore>
f 0 3600 ;Dummy f-table
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*OSCsend, OSCinit*

More information on this opcode: [http://www.youtube.com/watch?v=JX1C3TqP\\_9Y](http://www.youtube.com/watch?v=JX1C3TqP_9Y) , made by Andrés Cabrera

## Credits

Author: John ffitch  
2005

Examples by: David Akbari, Andrés Cabrera and Jonathan Murphy 2007

types aAG are new in Csound 6.07

# OSCCraw

OSCCraw — Listen for all OSC messages at a given port.

## Description

On each k-cycle looks to see if an OSC message has been received at a given port and copies its contents to a string array. All messages are copied. If a bundle of messages is received, the output array will contain all of the messages in it.

## Syntax

```
Smess[],klen OSCCraw iport
```

## Initialization

*iport* -- a port where messages are going to be received from.

## Performance

*Smess[]* -- an array of strings containing the message components: address, types, and data items. If the array does not exist (ie. has not been initialised), one will be created with size 2. Arrays cannot grow with the size of the message, so if a message with more than 2 items (ie. with any data items) is to be fully copied into the output, a larger array needs to be initialised before it is used. If a message has more items than an array has space for, it is truncated.

*klen* -- number of items placed in the output array. It is 0 if no message has been received, and at least 2 if a message has been received (address, types are the minimum items).

### Example 629. Example.

The following CSD file demonstrates how a message is received by OSCraw:

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

instr 1
  kwhen init 0
  kmetro metro 1
  kwhen += kmetro
  OSCsend kwhen, "127.0.0.1",7771, "/foo/bar", "f", kwhen
endin

instr 2
Smess[] init 3
top:
Smess,ka OSCCraw 7771
kn = 0
while kn < ka do
  printf " : %s ", kn+1, Smess[kn]
  kn += 1
```

```
od
printf "%d items\n", ka, kn
if ka > 0 kgoto top
endin

</CsInstruments>
<CsScore>
i1 0 10
i2 0 10
</CsScore>
</CsoundSynthesizer>
```

## See Also

*OSCsend, OSCinit OSClisten*

## Credits

Authors: Oeyvind Brandtsegg, Victor Lazzarini  
2017

types aAG are new in Csound 6.07

# OSCsend

OSCsend — Sends data to other processes using the OSC protocol

## Description

Uses the OSC protocol to send message to other OSC listening processes.

## Syntax

```
OSCsend kwhen, ihost, iport, idestination, itype [, xdata1, xdata2, ...]
```

## Initialization

*ihost* -- a string that is the intended host computer domain name. An empty string is interpreted as the current computer.

*iport* -- the number of the port that is used for the communication.

*idestination* -- a string that is the destination address. This takes the form of a file name with directories. Csound just passes this string to the raw sending code and makes no interpretation.

*itype* -- a string that indicates the types of the optional arguments that are read at k-rate. The string can contain the characters "abcdfilmstAG" which stand for audio, Boolean, character, double, float, 32-bit integer, 64-bit integer, MIDI, string, timestamp, k-rate array and ftable.

## Performance

*kwhen* -- a message is sent whenever this value changes. A message will always be sent on the first call.

The data is taken from the k-values or a-value that follow the format string. In a similar way to a printf format, the characters in order determine how the argument is interpreted. Note that a time stamp takes two arguments.

## Example

The example shows a simple instrument, which when called, sends a group of 3 messages to a computer called "xenakis", on port 7770, to be read by a process that recognises /foo/bar as its address.

```
instr 1
  OSCsend 1, "xenakis.cs.bath.ac.uk", 7770, "/foo/bar", "sis", "FOO", 42, "bar"
endin
```

See the entry for *OSClisten*, for an example of send/ receive usage using OSC.

## See Also

*OSClisten*, *OSCinit*

More information on this opcode: [http://www.youtube.com/watch?v=JX1C3TqP\\_9Y](http://www.youtube.com/watch?v=JX1C3TqP_9Y) , made by Andrés Cabrera

## Credits

Author: John ffitch  
2005

Data types a, A and G were added in 6.07; they send data via blobs and are only really useful to other Csound instances or custom listeners.

## out32

`out32` — Writes 32-channel audio data to an external device or stream.

### Description

Writes 32-channel audio data to an external device or stream.

### Syntax

```
out32 asig1, asig2, asig3, asig4, asig5, asig6, asig7, asig8, asig10, \  
      asig11, asig12, asig13, asig14, asig15, asig16, asig17, asig18, \  
      asig19, asig20, asig21, asig22, asig23, asig24, asig25, asig26, \  
      asig27, asig28, asig29, asig30, asig31, asig32
```

### Performance

`out32` outputs 32 channels of audio.

### See Also

*outc, outch, outx, outz*

### Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2000

New in Csound Version 4.07

# out

out — Writes audio data to an external device or stream.

## Description

Writes audio data to an external device or stream, either from audio variables or from an audio array.

## Syntax

```
out asig1[, asig2,...]
out aarray
```

## Performance

Sends audio samples to an accumulating output buffer (created at the beginning of performance) which serves to collect the output of all active instruments before the sound is written to disk. There can be any number of these output units in an instrument.

In the array case each element of the array is sent to the corresponding channel. This provides a way of using many channels. The nonarray version is limited to 1999.

The type (mono, stereo, quad, hex, or oct) should agree with *nchnls*. But as of version 3.50, Csound will attempt to change an incorrect opcode to agree with *nchnls* statement.

## Examples

Here is an example of the out opcode. It uses the file *out.csd* [examples/out.csd].

### Example 630. Example of the out opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o out.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

instr 1

kamp = .6
kcps = 440
ifn = p4
```

```
asig oscil kamp, kcps, ifn
      out asig ;one channel

endin
</CsInstruments>
<CsScore>
f1 0 16384 10 1                               ; Sine
f2 0 16384 10 1 0.5 0.3 0.25 0.2 0.167 0.14 0.125 .111 ; Sawtooth

i 1 0 2 1
i 1 3 2 2

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*outh, outo, outq, outq1, outq2, outq3, outq4, outs, outs1, outs2, soundout*

## Credits

Author: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997  
Author: John ffitch  
NUIM, 2013

Original in Csound v1

Aray version in 6.01

# outc

outc — Writes audio data with an arbitrary number of channels to an external device or stream.

## Description

Writes audio data with an arbitrary number of channels to an external device or stream.

## Syntax

```
outc asig1 [, asig2] [...]
```

## Performance

*outc* outputs as many channels as provided. Any channels greater than *nchnls* are ignored. Zeros are added as necessary

## Examples

Here is an example of the outc opcode. It uses the file *outc.csd* [examples/outc.csd].

### Example 631. Example of the outc opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o outc.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 5
0dbfs = 1

instr 1

asig vco2 .05, 30 ; sawtooth waveform at low volume

kcut line 100, p3, 30 ; Vary cutoff frequency
kresonance = 7
inumlayer = 2
asig lowresx asig, kcut, kresonance, inumlayer
; output same sound to 5 channels
    outc asig,asig,asig,asig,asig

endin
</CsInstruments>
<CsScore>
```

```
i 1 0 30  
e  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*out32, outch, outx, outz*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2000

New in Csound Version 4.07

# outch

outch — Writes multi-channel audio data, with user-controllable channels, to an external device or stream.

## Description

Writes multi-channel audio data, with user-controllable channels, to an external device or stream.

## Syntax

```
outch kchan1, asig1 [, kchan2] [, asig2] [...]
```

## Performance

*outch* outputs *asig1* on the channel determined by *kchan1*, *asig2* on the channel determined by *kchan2*, etc.



### Note

The highest number for *kchanX* available for use with *outch* depends on *nchnls*. If *kchanX* is greater than *nchnls*, *asigX* will be silent. Note that *outch* will give a warning but not an error in this case.

## Examples

Here is an example of the outch opcode. It uses the file *outch.csd* [examples/outch.csd].

### Example 632. Example of the outch opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o outch.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 5
0dbfs = 1

instr 1

asig vco2 .05, 100 ; sawtooth waveform at low volume

kcut line 100, p3, 30 ; Vary cutoff frequency
kresonance = .7
inumlayer = 3
asig lowresx asig, kcut, kresonance, inumlayer
```

```

klfo lfo 4, .5, 4
klfo = klfo+1 ; offset of 1
printks "signal is sent to channel %d\\n", .1, klfo
      outch klfo,asig

endin
</CsInstruments>
<CsScore>

i 1 0 30
e
</CsScore>
</CsoundSynthesizer>

```

```

signal is sent to channel 5
signal is sent to channel 4
signal is sent to channel 3
signal is sent to channel 2
signal is sent to channel 1
signal is sent to channel 5
.....

```

Here is another example of the outch opcode. It uses the file *outch-2.csd* [examples/outch.csd].

### Example 633. Another example of the outch opcode.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o outch-2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 4
0dbfs = 1

seed      0

instr 1 ;random movements between 4 speakers with outch

ichn1     random    1, 4.999 ;channel to start
ichn2     random    1, 4.999 ;channel to end
prints    "Moving from speaker %d to speaker %d\\n", int(ichn1), int(ichn2)
asamp     soundin   "fox.wav"
kmov      linseg    0, p3, 1
a1, a2    pan2      asamp, kmov
          outch     int(ichn1), a1, int(ichn2), a2

endin

</CsInstruments>
<CsScore>
r 5
i 1 0 3
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*out32, outc, outx, outz*

More information on this opcode: <http://www.csoundjournal.com/issue16/audiorouting.html> , written by Andreas Russo

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2000

New in Csound Version 4.07

# outh

outh — Writes 6-channel audio data to an external device or stream.

## Description

Writes 6-channel audio data to an external device or stream.

## Syntax

```
outh asig1, asig2, asig3, asig4, asig5, asig6
```

## Performance

Sends 6-channel audio samples to an accumulating output buffer (created at the beginning of performance) which serves to collect the output of all active instruments before the sound is written to disk. There can be any number of these output units in an instrument.

The type (mono, stereo, quad, hex, or oct) should agree with *nchnls*. But as of version 3.50, Csound will attempt to change an incorrect opcode to agree with *nchnls* statement.

## See Also

*out, outo, outq, outq1, outq2, outq3, outq4, outs, outs1, outs2, soundout*

## Credits

Author: John fitch

Introduced before Version 3

# outiat

outiat — Sends MIDI aftertouch messages at i-rate.

## Description

Sends MIDI aftertouch messages at i-rate.

## Syntax

```
outiat ichn, ivalue, imin, imax
```

## Initialization

*ichn* -- MIDI channel number (1-16)

*ivalue* -- floating point value

*imin* -- minimum floating point value (converted in MIDI integer value 0)

*imax* -- maximum floating point value (converted in MIDI integer value 127 (7 bit))

## Performance

*outiat* (i-rate aftertouch output) sends aftertouch messages. It works only with MIDI instruments which recognize them. It can drive a different value of a parameter for each note currently active.

It can scale an i-value floating-point argument according to the *imin* and *imax* values. For example, set *imin* = 1.0 and *imax* = 2.0. When the *ivalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *ivalue* argument receives a 1.0 value, it will send a 0 value. i-rate opcodes send their message once during instrument initialization.

## Examples

Here is an example of the *outiat* opcode. It uses the file *outiat.csd* [examples/outiat.csd].

### Example 634. Example of the *outiat* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -Q1 -M0 ;;realtime audio out and midi in and out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o outiat.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 10
```

```
nchnls = 2

instr 1

ikey notnum
ivel veloc

ivib = 25 ;low value.
outiat 1, ivib, 0, 127 ;assign aftertouch on
print ivib ;external synth for example to
midion 1, ikey, ivel ;change depth of filter modulation

endin
</CsInstruments>
<CsScore>
f0 30 ;play for 30 seconds

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*outic14, outic, outipat, outipb, outipc, outkat, outkc14, outkc, outkpat, outkpb, outkpc*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# outic14

`outic14` — Sends 14-bit MIDI controller output at i-rate.

## Description

Sends 14-bit MIDI controller output at i-rate.

## Syntax

```
outic14 ichn, imsb, ilsb, ivalue, imin, imax
```

## Initialization

*ichn* -- MIDI channel number (1-16)

*imsb* -- most significant byte controller number when using 14-bit parameters (0-127)

*ilsb* -- least significant byte controller number when using 14-bit parameters (0-127)

*ivalue* -- floating point value

*imin* -- minimum floating point value (converted in MIDI integer value 0)

*imax* -- maximum floating point value (converted in MIDI integer value 16383 (14-bit))

## Performance

*outic14* (i-rate MIDI 14-bit controller output) sends a pair of controller messages. This opcode can drive 14-bit parameters on MIDI instruments that recognize them. The first control message contains the most significant byte of *ivalue* argument while the second message contains the less significant byte. *imsb* and *ilsb* are the number of the most and less significant controller.

This opcode can drive a different value of a parameter for each note currently active.

It can scale an i-value floating-point argument according to the *imin* and *imax* values. For example, set *imin* = 1.0 and *imax* = 2.0. When the *ivalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *ivalue* argument receives a 1.0 value, it will send a 0 value. i-rate opcodes send their message once during instrument initialization.

## See Also

*outiat*, *outic*, *outipat*, *outipb*, *outipc*, *outkat*, *outkc14*, *outkc*, *outkpat*, *outkpb*, *outkpc*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# outic

outic — Sends MIDI controller output at i-rate.

## Description

Sends MIDI controller output at i-rate.

## Syntax

```
outic ichn, inum, ivalue, imin, imax
```

## Initialization

*ichn* -- MIDI channel number (1-16)

*inum* -- controller number (0-127 for example 1 = ModWheel; 2 = BreathControl etc.)

*ivalue* -- floating point value

*imin* -- minimum floating point value (converted in MIDI integer value 0)

*imax* -- maximum floating point value (converted in MIDI integer value 127 (7 bit))

## Performance

*outic* (i-rate MIDI controller output) sends controller messages to the MIDI OUT device. It works only with MIDI instruments which recognize them. It can drive a different value of a parameter for each note currently active.

It can scale an i-value floating-point argument according to the *imin* and *imax* values. For example, set *imin* = 1.0 and *imax* = 2.0. When the *ivalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *ivalue* argument receives a 1.0 value, it will send a 0 value. i-rate opcodes send their message once during instrument initialization.

## Examples

Here is an example of the *outic* opcode. It uses the file *outic.csd* [examples/outic.csd].

### Example 635. Example of the *outic* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -Q1 -M0 ;;realtime audio out -+rtmidi=virtual
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o outic.wav -W ;; for file output any platform
</CsOptions>
```

```
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

instr 1

ikey notnum
ivel veloc
kbrt = 40 ;set controller 74 (=brightness)
outic 1, 74, kbrt, 0, 127 ;so filter closes a bit
midion 1, ikey, ivel ;play external synth

endin
</CsInstruments>
<CsScore>
f0 30 ;runs 30 seconds

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*outiat, outic14, outipat, outipb, outipc, outkat, outkc14, outkc, outkpat, outkpb, outkpc*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# outipat

outipat — Sends polyphonic MIDI aftertouch messages at i-rate.

## Description

Sends polyphonic MIDI aftertouch messages at i-rate.

## Syntax

```
outipat ichn, inotenum, ivalue, imin, imax
```

## Initialization

*ichn* -- MIDI channel number (1-16)

*inotenum* -- MIDI note number (used in polyphonic aftertouch messages)

*ivalue* -- floating point value

*imin* -- minimum floating point value (converted in MIDI integer value 0)

*imax* -- maximum floating point value (converted in MIDI integer value 127 (7 bit))

## Performance

*outipat* (i-rate polyphonic aftertouch output) sends polyphonic aftertouch messages. It works only with MIDI instruments which recognize them. It can drive a different value of a parameter for each note currently active.

It can scale an i-value floating-point argument according to the *imin* and *imax* values. For example, set *imin* = 1.0 and *imax* = 2.0. When the *ivalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *ivalue* argument receives a 1.0 value, it will send a 0 value. i-rate opcodes send their message once during instrument initialization.

## See Also

*outiat*, *outic14*, *outic*, *outipb*, *outipc*, *outkat*, *outkc14*, *outkc*, *outkpat*, *outkpb*, *outkpc*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# outipb

outipb — Sends MIDI pitch-bend messages at i-rate.

## Description

Sends MIDI pitch-bend messages at i-rate.

## Syntax

```
outipb ichn, ivalue, imin, imax
```

## Initialization

*ichn* -- MIDI channel number (1-16)

*ivalue* -- floating point value

*imin* -- minimum floating point value (converted in MIDI integer value 0)

*imax* -- maximum floating point value (converted in MIDI integer value 127 (7 bit))

## Performance

*outipb* (i-rate pitch bend output) sends pitch bend messages. It works only with MIDI instruments which recognize them. It can drive a different value of a parameter for each note currently active.

It can scale an i-value floating-point argument according to the *imin* and *imax* values. For example, set *imin* = 1.0 and *imax* = 2.0. When the *ivalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *ivalue* argument receives a 1.0 value, it will send a 0 value. i-rate opcodes send their message once during instrument initialization.

## Examples

Here is an example of the outipb opcode. It uses the file *outipb.csd* [examples/outipb.csd].

### Example 636. Example of the outipb opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -Q1 -M0 ;;realtime audio out and midi in and out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o outipb.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 10
```

```
nchnls = 2

instr 1

ikey notnum
ivel veloc

ipb = 10 ;a little out of tune
outipb 1, ipb, 0, 127 ;(= pitchbend)
midion 1, ikey, ivel ;of external synth

endin
</CsInstruments>
<CsScore>
f0 30

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*outiat, outic14, outic, outipat, outipc, outkat, outkc14, outkc, outkpat, outkpb, outkpc*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# outipc

outipc — Sends MIDI program change messages at i-rate

## Description

Sends MIDI program change messages at i-rate

## Syntax

```
outipc ichn, iprog, imin, imax
```

## Initialization

*ichn* -- MIDI channel number (1-16)

*iprog* -- program change number in floating point

*imin* -- minimum floating point value (converted in MIDI integer value 0)

*imax* -- maximum floating point value (converted in MIDI integer value 127 (7 bit))

## Performance

*outipc* (i-rate program change output) sends program change messages. It works only with MIDI instruments which recognize them. It can drive a different value of a parameter for each note currently active.

It can scale an i-value floating-point argument according to the *imin* and *imax* values. For example, set *imin* = 1.0 and *imax* = 2.0. When the *ivalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *ivalue* argument receives a 1.0 value, it will send a 0 value. i-rate opcodes send their message once during instrument initialization.

## Examples

Here is an example of the outipc opcode. It uses the file *outipc.csd* [examples/outipc.csd].

### Example 637. Example of the outipc opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -Q1 -M0 ;;realtime audio out and midi in and out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o outipc.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
```

```
nchnls = 2

instr 1

outipc 1, 80, 0, 127 ;program change --> 80
ikey notnum
ivel veloc
midion 1, ikey, ivel ;play external synth

endin
</CsInstruments>
<CsScore>
f0 30 ;runs 30 seconds

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*outiat, outic14, outic, outipat, outipb, outkat, outkc14, outkc, outkpat, outkpb, outkpc*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# outkat

outkat — Sends MIDI aftertouch messages at k-rate.

## Description

Sends MIDI aftertouch messages at k-rate.

## Syntax

```
outkat kchn, kvalue, kmin, kmax
```

## Performance

*kchn* -- MIDI channel number (1-16)

*kvalue* -- floating point value

*kmin* -- minimum floating point value (converted in MIDI integer value 0)

*kmax* -- maximum floating point value (converted in MIDI integer value 127)

*outkat* (k-rate aftertouch output) sends aftertouch messages. It works only with MIDI instruments which recognize them. It can drive a different value of a parameter for each note currently active.

It can scale the k-value floating-point argument according to the *kmin* and *kmax* values. For example: set *kmin* = 1.0 and *kmax* = 2.0. When the *kvalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *kvalue* argument receives a 1.0 value, it will send a 0 value. k-rate opcodes send a message each time the MIDI converted value of argument *kvalue* changes.

## Examples

Here is an example of the outkat opcode. It uses the file *outkat.csd* [examples/outkat.csd].

### Example 638. Example of the outkat opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -Q1 -M0 ;;realtime audio out and midi in and out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o outkat.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 10
nchnls = 2

instr 1
```

```
ikey notnum
ivel veloc

kvib linseg 100, .5, 120 ;vary aftertouch in .5 second
kvbr = int(kvib) ;whole numbers only
outkat 1, kvbr, 0, 127 ;assign aftertouch on
printk2 kvbr ;external synth for example to
midion 1, ikey, ivel ;change depth of filter modulation

endin
</CsInstruments>
<CsScore>
f0 30 ;play for 30 seconds

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*outiat, outic14, outic, outipat, outipb, outipc, outkc14, outkc, outkpat, outkpb, outkpc*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# outkc14

outkc14 — Sends 14-bit MIDI controller output at k-rate.

## Description

Sends 14-bit MIDI controller output at k-rate.

## Syntax

```
outkc14 kchn, kmsb, klsb, kvalue, kmin, kmax
```

## Performance

*kchn* -- MIDI channel number (1-16)

*kmsb* -- most significant byte controller number when using 14-bit parameters (0-127)

*klsb* -- least significant byte controller number when using 14-bit parameters (0-127)

*kvalue* -- floating point value

*kmin* -- minimum floating point value (converted in MIDI integer value 0)

*kmax* -- maximum floating point value (converted in MIDI integer value 16383 (14-bit))

*outkc14* (k-rate MIDI 14-bit controller output) sends a pair of controller messages. It works only with MIDI instruments which recognize them. These opcodes can drive 14-bit parameters on MIDI instruments that recognize them. The first control message contains the most significant byte of *kvalue* argument while the second message contains the less significant byte. *kmsb* and *klsb* are the number of the most and less significant controller.

It can drive a different value of a parameter for each note currently active.

It can scale the k-value floating-point argument according to the *kmin* and *kmax* values. For example: set *kmin* = 1.0 and *kmax* = 2.0. When the *kvalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *kvalue* argument receives a 1.0 value, it will send a 0 value. k-rate opcodes send a message each time the MIDI converted value of argument *kvalue* changes.

## See Also

*outiat*, *outic14*, *outic*, *outipat*, *outipb*, *outipc*, *outkat*, *outkc*, *outkpat*, *outkpb*, *outkpc*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# outkc

outkc — Sends MIDI controller messages at k-rate.

## Description

Sends MIDI controller messages at k-rate.

## Syntax

`outkc kchn, knum, kvalue, kmin, kmax`

## Performance

*kchn* -- MIDI channel number (1-16)

*knun* -- controller number (0-127 for example 1 = ModWheel; 2 = BreathControl etc.)

*kvalue* -- floating point value

*kmin* -- minimum floating point value (converted in MIDI integer value 0)

*kmax* -- maximum floating point value (converted in MIDI integer value 127 (7 bit))

*outkc* (k-rate MIDI controller output) sends controller messages to MIDI OUT device. It works only with MIDI instruments which recognize them. It can drive a different value of a parameter for each note currently active.

It can scale the k-value floating-point argument according to the *kmin* and *kmax* values. For example: set *kmin* = 1.0 and *kmax* = 2.0. When the *kvalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *kvalue* argument receives a 1.0 value, it will send a 0 value. k-rate opcodes send a message each time the MIDI converted value of argument *kvalue* changes.

## Examples

Here is an example of the outkc opcode. It uses the file *outkc.csd* [examples/outkc.csd].

### Example 639. Example of the outkc opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -Q1 -M0   ;;realtime audio out and midi in and out
;-iadc          ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o outkc.wav -W   ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
```

```
ksmps = 32
nchnls = 2

instr 1

ikey notnum
ivel veloc

kcut linseg 100, .5, 20 ;vary controller in .5 second
kbrt = int(kcut) ;whole numbers only
outkc 1, 74, kbrt, 0, 127 ;controller 74 (= brightness)
midion 1, ikey, ivel ;of external synth

endin
</CsInstruments>
<CsScore>
f0 30 ;runs 30 seconds

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*outiat, outic14, outic, outipat, outipb, outipc, outkat, outkc14, outkpat, outkpb, outkpc*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# outkpat

outkpat — Sends polyphonic MIDI aftertouch messages at k-rate.

## Description

Sends polyphonic MIDI aftertouch messages at k-rate.

## Syntax

```
outkpat kchn, knotenum, kvalue, kmin, kmax
```

## Performance

*kchn* -- MIDI channel number (1-16)

*knotenum* -- MIDI note number (used in polyphonic aftertouch messages)

*kvalue* -- floating point value

*kmin* -- minimum floating point value (converted in MIDI integer value 0)

*kmax* -- maximum floating point value (converted in MIDI integer value 127 (7 bit))

*outkpat* (k-rate polyphonic aftertouch output) sends polyphonic aftertouch messages. It works only with MIDI instruments which recognize them. It can drive a different value of a parameter for each note currently active.

It can scale the k-value floating-point argument according to the *kmin* and *kmax* values. For example: set *kmin* = 1.0 and *kmax* = 2.0. When the *kvalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *kvalue* argument receives a 1.0 value, it will send a 0 value. k-rate opcodes send a message each time the MIDI converted value of argument *kvalue* changes.

## See Also

*outiat*, *outic14*, *outic*, *outipat*, *outipb*, *outipc*, *outkat*, *outkc14*, *outkc*, *outkpb*, *outkpc*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# outkpb

outkpb — Sends MIDI pitch-bend messages at k-rate.

## Description

Sends MIDI pitch-bend messages at k-rate.

## Syntax

```
outkpb kchn, kvalue, kmin, kmax
```

## Performance

*kchn* -- MIDI channel number (1-16)

*kvalue* -- floating point value

*kmin* -- minimum floating point value (converted in MIDI integer value 0)

*kmax* -- maximum floating point value (converted in MIDI integer value 127 (7 bit))

*outkpb* (k-rate pitch-bend output) sends pitch-bend messages. It works only with MIDI instruments which recognize them. It can drive a different value of a parameter for each note currently active.

It can scale the k-value floating-point argument according to the *kmin* and *kmax* values. For example: set *kmin* = 1.0 and *kmax* = 2.0. When the *kvalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *kvalue* argument receives a 1.0 value, it will send a 0 value. k-rate opcodes send a message each time the MIDI converted value of argument *kvalue* changes.

## Examples

Here is an example of the outkpb opcode. It uses the file *outkpb.csd* [examples/outkpb.csd].

### Example 640. Example of the outkpb opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -Q1 -M0 ;;realtime audio out and midi in and out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o outkpb.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 10
nchnls = 2

instr 1
```

```
ikey notnum
ivel veloc

kpch linseg 100, 1, 0 ;vary in 1 second
kpb = int(kpch) ;whole numbers only
outkpb 1, kpb, 0, 127 ;(= pitchbend)
midion 1, ikey, ivel ;of external synth

endin
</CsInstruments>
<CsScore>
f0 30

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*outiat, outic14, outic, outipat, outipb, outipc, outkat, outkc14, outkc, outkpat, outkpc*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# outkpc

outkpc — Sends MIDI program change messages at k-rate.

## Description

Sends MIDI program change messages at k-rate.

## Syntax

```
outkpc kchn, kprog, kmin, kmax
```

## Performance

*kchn* -- MIDI channel number (1-16)

*kprog* -- program change number in floating point

*kmin* -- minimum floating point value (converted in MIDI integer value 0)

*kmax* -- maximum floating point value (converted in MIDI integer value 127 (7 bit))

*outkpc* (k-rate program change output) sends program change messages. It works only with MIDI instruments which recognize them. These opcodes can drive a different value of a parameter for each note currently active.

It can scale the k-value floating-point argument according to the *kmin* and *kmax* values. For example: set *kmin* = 1.0 and *kmax* = 2.0. When the *kvalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *kvalue* argument receives a 1.0 value, it will send a 0 value. k-rate opcodes send a message each time the MIDI converted value of argument *kvalue* changes.

## Examples

Here is an example of the *outkpc* opcode. It uses the file *outkpc.csd* [examples/outkpc.csd].

### Example 641. Example of the *outkpc* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

This example generates a program change and a note on Csound's MIDI output port whenever a note is received on channel 1. Be sure to have something connected to Csound's MIDI out port to hear the result.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc      -d          -M0  -Q1  ;;RT audio I/O with MIDI in
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
```

```

ksmps = 10
nchnls = 2

; Example by Giorgio Zucco 2007

kprogram init 0

instr 1 ;Triggered by MIDI notes on channel 1

    ifund    notnum
    ivel    veloc
    idur = 1

; Sends a MIDI program change message according to
; the triggering note's velocity
outkpc      1 ,ivel ,0 ,127

noteondur   1 ,ifund ,ivel ,idur

endin

</CsInstruments>
<CsScore>
; Dummy ftable
f 0 60
</CsScore>
</CsoundSynthesizer>

```

Here is another example of the outkpc opcode. It uses the file *outkpc\_flkt.csd* [examples/outkpc\_flkt.csd].

### Example 642. Example of the outkpc opcode using FLTK.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc      -d          -M0  -Q1;;;RT audio I/O with MIDI in
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

; Example by Giorgio Zucco 2007

FLpanel "outkpc",200,100,90,90;start of container
gkpg, gihandle FLcount "Midi-Program change",0,127,1,5,1,152,40,16,23,-1
FLpanelEnd

FLrun

instr 1

ktrig changed gkpg
outkpc      ktrig,gkpg,0,127

endin

</CsInstruments>

```

```
<CsScore>  
; Run instrument 1 for 60 seconds  
i 1 0 60  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*outiat, outic14, outic, outipat, outipb, outipc, outkat, outkc14, outkc, outkpat, outkpb*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# outleta

outleta — Sends an arate signal out from an instrument to a named port.

## Description

Sends an arate signal out from an instrument to a named port.

## Syntax

```
outleta Sname, asignal
```

## Initialization

*Sname* -- String name of the outlet port. The name of the outlet is implicitly qualified by the instrument name or number, so it is valid to use the same outlet name in more than one instrument (but not to use the same outlet name twice in one instrument).

## Performance

*asignal* -- audio output signal

During performance, the audio output signal is sent to each instance of an instrument containing an inlet port to which this outlet has been connected using the connect opcode. The signals of all the outlets connected to an inlet are summed in the inlet.

## Examples

Here is an example of the outleta opcode. It uses the file *outleta.csd* [examples/outleta.csd].

### Example 643. Example of the outleta opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o outleta.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

connect "1", "Outl", "reverby", "InL"
connect "1", "Outr", "reverby", "InR"
```

```
alwayson "reverby", 1

instr 1

aIn diskIn2 "fox.wav", 1
  outleta "Outl", aIn
  outleta "Outr", aIn

endin

instr reverby

aInL   inleta "InL"
aInR   inleta "InR"

al, ar reverbSc aInL, aInR, 0.7, 21000
ifxlev = 0.5
al     = (aInL*ifxlev)+(al*(1-ifxlev))
ar     = (aInR*ifxlev)+(ar*(1-ifxlev))
outs  al, ar

endin
</CsInstruments>
<CsScore>

i 1 0 3
e4
</CsScore>
</CsoundSynthesizer>
```

## See Also

*outletk outletkid outletf inleta inletk inletkid inletf connect alwayson ftgenonce*

More information on this opcode: <http://www.csoundjournal.com/issue13/signalFlowGraphOpcodes.html>  
, written by Michael Gogins

## Credits

By: Michael Gogins 2009

# outletf

outletf — Sends a frate signal (fsig) out from an instrument to a named port.

## Description

Sends a frate signal (fsig) out from an instrument to a named port.

## Syntax

```
outletf Sname, fsignal
```

## Initialization

*Sname* -- String name of the outlet port. The name of the outlet is implicitly qualified by the instrument name or number, so it is valid to use the same outlet name in more than one instrument (but not to use the same outlet name twice in one instrument).

## Performance

*fsignal* -- frate output signal (fsig)

During performance, the output signal is sent to each instance of an instrument containing an inlet port to which this outlet has been connected using the connect opcode. The signals of all the outlets connected to an inlet are combined in the inlet.

## See Also

*outleta outletk outletkid inleta inleth inletkid inletf connect alwayson ftgenonce*

## Credits

By: Michael Gogins 2009

# outletk

outletk — Sends a krate signal out from an instrument to a named port.

## Description

Sends a krate signal out from an instrument to a named port.

## Syntax

```
outletk Sname, ksignal
```

## Initialization

*Sname* -- String name of the outlet port. The name of the outlet is implicitly qualified by the instrument name or number, so it is valid to use the same outlet name in more than one instrument (but not to use the same outlet name twice in one instrument).

## Performance

*ksignal* -- krate output signal

During performance, the krate output signal is sent to each instance of an instrument containing an inlet port to which this outlet has been connected using the connect opcode. The signals of all the outlets connected to an inlet are summed in the inlet.

## Examples

Here is an example of the outletk opcode. It uses the file *outletk.csd* [examples/outletk.csd].

### Example 644. Example of the outletk opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o inletk.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

connect "bend", "bendout", "guitar", "bendin"

instr bend
```

```
kbend line p4, p3, p5
      outletk "bendout", kbend
endin

instr guitar

kbend inletk "bendin"
kpch pow 2, kbend/12
      printk2 kpch
asig oscili .4, 440*kpch, 1
      outs asig, asig
endin

</CsInstruments>
<CsScore>
f1 0 1024 10 1

i"guitar" 0 5 8.00
i"bend" 3 .2 -12 12
i"bend" 4 .1 -17 40
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*outleta outletf outletkid inleta inletk inletkid inletf connect alwayson ftgenonce*

More information on this opcode: <http://www.csoundjournal.com/issue13/signalFlowGraphOpcodes.html>  
, written by Michael Gogins

## Credits

By: Michael Gogins 2009

# outletkid

outletkid — Sends a krate signal out from an instrument to a named port.

## Description

Sends a krate signal out from an instrument to a named port.

## Syntax

```
outletkid Sname, SinstanceID, ksignal
```

## Initialization

*Sname* -- String name of the outlet port. The name of the outlet is implicitly qualified by the instrument name or number, so it is valid to use the same outlet name in more than one instrument (but not to use the same outlet name twice in one instrument).

*SinstanceID* -- String name of the outlet port's instance ID. This enables the inlet to discriminate between different instances of the outlet, e.g. one instance of the outlet might be created by a note specifying one instance ID, and another instance might be created by a note specifying another ID. This might be used, e.g., to situate difference instances of an instrument at different points in an Ambisonic space in a spatializing effects processor.

## Performance

*ksignal* -- krate output signal

During performance, the krate output signal is sent to each instance of an instrument containing an inlet port to which this outlet has been connected using the connect opcode. The signals of all the outlets connected to an inlet, but only those that have a matching instance ID, are summed in the inlet.

## See Also

*outleta outletf inleta inleth inletkid inletf connect alwayson ftgenonce*

More information on this opcode: <http://www.csoundjournal.com/issue13/signalFlowGraphOpcodes.html>, written by Michael Gogins

## Credits

By: Michael Gogins 2009

# outletv

outletv — Sends an arate array signal out from an instrument to a named port.

## Description

Sends an arate array signal out from an instrument to a named port.

## Syntax

```
outletv Sname, array
```

## Initialization

*Sname* -- String name of the outlet port. The name of the outlet is implicitly qualified by the instrument name or number, so it is valid to use the same outlet name in more than one instrument (but not to use the same outlet name twice in one instrument).

## Performance

*array* -- arate array output signal

During performance, the arate array output signal is sent to each instance of an instrument containing an inlet port to which this outlet has been connected using the connect opcode. The signals of all the outlets connected to an inlet are summed in the inlet. The ports may have any number of channels, but the inlet port must have the same number of channels as the connected outlet ports.

## See Also

*outleta outletf outletk outletkid inleta inletf inletk inletkid inletv connect alwayson ftgenonce*

More information on this opcode: <http://www.csoundjournal.com/issue13/signalFlowGraphOpcodes.html>, written by Michael Gogins

## Credits

By: Michael Gogins 2009

## outo

`outo` — Writes 8-channel audio data to an external device or stream.

### Description

Writes 8-channel audio data to an external device or stream.

### Syntax

```
outo asig1, asig2, asig3, asig4, asig5, asig6, asig7, asig8
```

### Performance

Sends 8-channel audio samples to an accumulating output buffer (created at the beginning of performance) which serves to collect the output of all active instruments before the sound is written to disk. There can be any number of these output units in an instrument.

The type (mono, stereo, quad, hex, or oct) should agree with *nchnls*. But as of version 3.50, Csound will attempt to change an incorrect opcode to agree with *nchnls* statement.

### See Also

*out, outh, outq, outq1, outq2, outq3, outq4, outs, outs1, outs2, soundout*

### Credits

Author: John ffitth

New after 3.30

# outq1

outq1 — Writes samples to quad channel 1 of an external device or stream.

## Description

Writes samples to quad channel 1 of an external device or stream.

## Syntax

```
outq1 asig
```

## Performance

Sends audio samples to an accumulating output buffer (created at the beginning of performance) which serves to collect the output of all active instruments before the sound is written to disk. There can be any number of these output units in an instrument.

The type (mono, stereo, quad, hex, or oct) should agree with *nchnls*. But as of version 3.50, Csound will attempt to change an incorrect opcode to agree with the *nchnls* statement. Opcodes can be chosen to direct sound to any particular channel: *outs1* sends to stereo channel 1, *outq3* to quad channel 3, etc.

## Examples

Here is an example of the outq1 opcode. It uses the file *outq1.csd* [examples/outq1.csd].

### Example 645. Example of the outq1 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o outq1.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 4
0dbfs = 1

instr 1

asig vco2 .05, 30 ; sawtooth waveform at low volume

kcut line 60, p3, 300 ; Vary cutoff frequency
kresonance = 7
inumlayer = 2
asig lowresx asig, kcut, kresonance, inumlayer
```

```
    outq1 asig ; output channel 1

endin
</CsInstruments>
<CsScore>

i 1 0 3
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*out, outh, outo, outq, outq2, outq3, outq4, outs, outs1, outs2, soundout*

## Credits

Author: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

# outq2

outq2 — Writes samples to quad channel 2 of an external device or stream.

## Description

Writes samples to quad channel 2 of an external device or stream.

## Syntax

```
outq2 asig
```

## Performance

Sends audio samples to an accumulating output buffer (created at the beginning of performance) which serves to collect the output of all active instruments before the sound is written to disk. There can be any number of these output units in an instrument.

The type (mono, stereo, quad, hex, or oct) should agree with *nchnls*. But as of version 3.50, Csound will attempt to change an incorrect opcode to agree with the *nchnls* statement. Opcodes can be chosen to direct sound to any particular channel: *outs1* sends to stereo channel 1, *outq3* to quad channel 3, etc.

## Examples

Here is an example of the outq2 opcode. It uses the file *outq2.csd* [examples/outq2.csd].

### Example 646. Example of the outq2 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o outq2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 4
0dbfs = 1

instr 1

asig vco2 .05, 30 ; sawtooth waveform at low volume

kcut line 300, p3, 60 ; Vary cutoff frequency
kresonance = 7
inumlayer = 2
asig lowresx asig, kcut, kresonance, inumlayer
```

```
    outq2 asig ; output channel 2

endin
</CsInstruments>
<CsScore>

i 1 0 3
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*out, outh, outo, outq, outq1, outq3, outq4, outs, outs1, outs2, soundout*

## Credits

Author: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

# outq3

outq3 — Writes samples to quad channel 3 of an external device or stream.

## Description

Writes samples to quad channel 3 of an external device or stream.

## Syntax

```
outq3 asig
```

## Performance

Sends audio samples to an accumulating output buffer (created at the beginning of performance) which serves to collect the output of all active instruments before the sound is written to disk. There can be any number of these output units in an instrument.

The type (mono, stereo, quad, hex, or oct) should agree with *nchnls*. But as of version 3.50, Csound will attempt to change an incorrect opcode to agree with the *nchnls* statement. Opcodes can be chosen to direct sound to any particular channel: *outs1* sends to stereo channel 1, *outq3* to quad channel 3, etc.

## Examples

Here is an example of the outq3 opcode. It uses the file *outq3.csd* [examples/outq3.csd].

### Example 647. Example of the outq3 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o outq3.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 4
0dbfs = 1

instr 1

asig vco2 .05, 30 ; sawtooth waveform at low volume

kcut line 30, p3, 100 ; Vary cutoff frequency
kresonance = 7
inumlayer = 2
asig lowresx asig, kcut, kresonance, inumlayer
```

```
    outq3 asig ; output channel 3

endin
</CsInstruments>
<CsScore>

i 1 0 3
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*out, outh, outo, outq, outq1, outq2, outq4, outs, outs1, outs2, soundout*

## Credits

Author: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

# outq4

outq4 — Writes samples to quad channel 4 of an external device or stream.

## Description

Writes samples to quad channel 4 of an external device or stream.

## Syntax

```
outq4 asig
```

## Performance

Sends audio samples to an accumulating output buffer (created at the beginning of performance) which serves to collect the output of all active instruments before the sound is written to disk. There can be any number of these output units in an instrument.

The type (mono, stereo, quad, hex, or oct) should agree with *nchnls*. But as of version 3.50, Csound will attempt to change an incorrect opcode to agree with the *nchnls* statement. Opcodes can be chosen to direct sound to any particular channel: *outs1* sends to stereo channel 1, *outq3* to quad channel 3, etc.

## Examples

Here is an example of the outq4 opcode. It uses the file *outq4.csd* [examples/outq4.csd].

### Example 648. Example of the outq4 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o outq4.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 4
0dbfs = 1

instr 1

asig vco2 .05, 30 ; sawtooth waveform at low volume

kcut line 100, p3, 30 ; Vary cutoff frequency
kresonance = 7
inumlayer = 2
asig lowresx asig, kcut, kresonance, inumlayer
```

```
    outq4 asig ; output channel 4

endin
</CsInstruments>
<CsScore>

i 1 0 3
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*out, outh, outo, outq, outq1, outq2, outq3, outs, outs1, outs2, soundout*

## Credits

Author: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

# outq

outq — Writes 4-channel audio data to an external device or stream.

## Description

Writes 4-channel audio data to an external device or stream.

## Syntax

```
outq asig1, asig2, asig3, asig4
```

## Performance

Sends 4-channel audio samples to an accumulating output buffer (created at the beginning of performance) which serves to collect the output of all active instruments before the sound is written to disk. There can be any number of these output units in an instrument.

The type (mono, stereo, quad, hex, or oct) should agree with *nchnls*. But as of version 3.50, Csound will attempt to change an incorrect opcode to agree with the *nchnls* statement. Opcodes can be chosen to direct sound to any particular channel: *outs1* sends to stereo channel 1, *outq3* to quad channel 3, etc.

## Examples

Here is an example of the outq opcode. It uses the file *outq.csd* [examples/outq.csd].

### Example 649. Example of the outq opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o outq.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 4
0dbfs = 1

instr 1

asig vco2 .01, 110 ; sawtooth waveform at low volume

;filter the first channel
kcut1 line 60, p3, 300 ; Vary cutoff frequency
kresonance1 = 3
inumlayer1 = 3
asig1 lowresx asig, kcut1, kresonance1, inumlayer1
```

```
;filter the second channel
kcut2 line 300, p3, 60 ; Vary cutoff frequency
kresonance2 = 3
inumlayer2 = 3
asig2 lowresx asig, kcut2, kresonance2, inumlayer2

;filter the third channel
kcut3 line 30, p3, 100; Vary cutoff frequency
kresonance3 = 6
inumlayer3 = 3
asig3 lowresx asig, kcut3, kresonance3, inumlayer3
asig3 = asig3*.1 ; lower volume

;filter the fourth channel
kcut4 line 100, p3, 30; Vary cutoff frequency
kresonance4 = 6
inumlayer4 = 3
asig4 lowresx asig, kcut4, kresonance4, inumlayer4
asig4 = asig4*.1 ; lower volume

    outq asig1, asig2, asig3, asig4; output channels 1, 2, 3 & 4

endin
</CsInstruments>
<CsScore>

i 1 0 5
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*out, outh, outo, outq1, outq2, outq3, outq4, outs, outs1, outs2, soundout*

## Credits

Author: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

# outrg

outrg — Allow output to a range of adjacent audio channels on the audio output device

## Description

*outrg* outputs audio to a range of adjacent audio channels on the audio output device.

## Syntax

```
outrg kstart, aout1 [,aout2, aout3, ..., aoutN]
```

## Performance

*kstart* - the number of the first channel of the output device to be accessed (channel numbers starts with 1, which is the first channel)

*aout1*, *aout2*, ... *aoutN* - the arguments containing the audio to be output to the corresponding output channels.

*outrg* allows to output a range of adjacent channels to the output device. *kstart* indicates the first channel to be accessed (channel 1 is the first channel). The user must be sure that the number obtained by summing *kstart* plus the number of accessed channels -1 is  $\leq nchnls$ .

## Examples

Here is an example of the *outrg* opcode. It uses the file *outrg.csd* [examples/outrg.csd].

### Example 650. Example of the *outrg* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o outrg.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 4      ;quad
0dbfs = 1

instr 1

kleft init 1
asig vco2 .5, 220 ;sawtooth
idur = p3/(nchnls-1)
knext init idur
kpos init 0
```

```
krate init 1/idur
kbase init 0
ktime timeinsts
if ktime>=knext then
  kleft = kleft + 1
  knext = knext + idur
  kpos = 0
  kbase = ktime
else
  kpos = (ktime-kbase)/idur
endif
printks "speaker %d position %f\n", 0, kleft, kpos
a1,a2 pan2 asig, kpos
  outrg kleft, a1, a2
kpos = kbase/idur
endin

</CsInstruments>
<CsScore>

i 1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like these:

```
speaker 1 position 0.000200
speaker 1 position 0.000400
speaker 1 position 0.000600
.....
speaker 1 position 1.000000
speaker 2 position 0.000000
speaker 2 position 0.000200
.....
speaker 2 position 0.999800
speaker 3 position 0.000000
speaker 3 position 0.000200
.....
speaker 3 position 0.999600
speaker 4 position 0.000000
.....
```

## Credits

Author: Gabriel Maldonado

New in version 5.06

# outs1

outs1 — Writes samples to stereo channel 1 of an external device or stream.

## Description

Writes samples to stereo channel 1 of an external device or stream.

## Syntax

```
outs1 asig
```

## Performance

Sends audio samples to an accumulating output buffer (created at the beginning of performance) which serves to collect the output of all active instruments before the sound is written to disk. There can be any number of these output units in an instrument.

The type (mono, stereo, quad, hex, or oct) should agree with *nchnls*. But as of version 3.50, Csound will attempt to change an incorrect opcode to agree with the *nchnls* statement. Opcodes can be chosen to direct sound to any particular channel: *outs1* sends to stereo channel 1, *outq3* to quad channel 3, etc.

## Examples

Here is an example of the *outs1* opcode. It uses the file *outs1.csd* [examples/outs1.csd].

### Example 651. Example of the *outs1* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o outs1.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

asig vco2 .01, 110 ; sawtooth waveform at low volume
kcut line 60, p3, 300 ; Vary cutoff frequency
kresonance = 3
inumlayer = 3
asig lowresx asig, kcut, kresonance, inumlayer
      outs1 asig ; output stereo channel 1 only
```

```
endin
</CsInstruments>
<CsScore>

i 1 0 3
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*out, outh, outo, outq, outq1, outq2, outq3, outq4, outs, outs2, soundout*

## Credits

Author: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

# outs2

outs2 — Writes samples to stereo channel 2 of an external device or stream.

## Description

Writes samples to stereo channel 2 of an external device or stream.

## Syntax

```
outs2 asig
```

## Performance

Sends audio samples to an accumulating output buffer (created at the beginning of performance) which serves to collect the output of all active instruments before the sound is written to disk. There can be any number of these output units in an instrument.

The type (mono, stereo, quad, hex, or oct) should agree with *nchnls*. But as of version 3.50, Csound will attempt to change an incorrect opcode to agree with the *nchnls* statement. Opcodes can be chosen to direct sound to any particular channel: *outs1* sends to stereo channel 1, *outq3* to quad channel 3, etc.

## Examples

Here is an example of the *outs2* opcode. It uses the file *outs2.csd* [examples/outs2.csd].

### Example 652. Example of the *outs2* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o outs2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

asig vco2 .01, 110 ; sawtooth waveform at low volume
kcut line 300, p3, 60 ; Vary cutoff frequency
kresonance = 3
inumlayer = 3
asig lowresx asig, kcut, kresonance, inumlayer
    outs2 asig ; output stereo channel 2 only
```

```
endin
</CsInstruments>
<CsScore>

i 1 0 3
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*out, outh, outo, outq, outq1, outq2, outq3, outq4, outs, outs1, soundout*

## Credits

Author: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

# outs

outs — Writes stereo audio data to an external device or stream.

## Description

Writes stereo audio data to an external device or stream.

## Syntax

```
outs asig1, asig2
```

## Performance

Sends stereo audio samples to an accumulating output buffer (created at the beginning of performance) which serves to collect the output of all active instruments before the sound is written to disk. There can be any number of these output units in an instrument.

The type (mono, stereo, quad, hex, or oct) should agree with *nchnls*. But as of version 3.50, Csound will attempt to change an incorrect opcode to agree with the *nchnls* statement. Opcodes can be chosen to direct sound to any particular channel: *outs1* sends to stereo channel 1, *outq3* to quad channel 3, etc.

## Examples

Here is an example of the outs opcode. It uses the file *outs.csd* [examples/outs.csd].

### Example 653. Example of the outs opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o outs.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

asig vco2 .01, 110 ; sawtooth waveform at low volume
;filter a channel
kcut1 line 60, p3, 300 ; Vary cutoff frequency
kresonance1 = 3
inumlayer1 = 3
asig1 lowresx asig, kcut1, kresonance1, inumlayer1
;filter the other channel
```

```
kcut2 line 300, p3, 60 ; Vary cutoff frequency
kresonance2 = 3
inumlayer2 = 3
asig2 lowresx asig, kcut2, kresonance2, inumlayer2

    outs asig1, asig2 ; output both channels 1 & 2

endin
</CsInstruments>
<CsScore>

i 1 0 3
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*out, outh, outho, outq, outq1, outq2, outq3, outq4, outs1, outs2, soundout*

## Credits

Author: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

# outvalue

outvalue — Sends an i-rate or k-rate signal or string to a user-defined channel.

## Description

Sends an irate or k-rate signal or string to a user-defined channel.

## Syntax

```
outvalue "channel name", ivalue
outvalue "channel name", kvalue
outvalue "channel name", "string"
```

## Performance

*"channel name"* -- An integer or string (in double-quotes) representing channel.

*ivalue, kvalue* -- The value that is sent to the channel.

*string* -- The string or string variable that is sent to the channel.

## Examples

Here is an example of the outvalue opcode. It uses the file *outvalue.csd* [examples/outvalue.csd].

### Example 654. Example of the outvalue opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
;run this example in CsoundQt, a Csound editor that provides widgets
;make the Widgets-panel visible, by clicking the Widgets symbol in the menu or pressing (Alt+1).

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
; written by Alex Hofmann

instr 1 ;move fader

kMoveUp linseg 0, 3, 1, 1, 1, 0.5, 0
outvalue "movefader", kMoveUp
endin

</CsInstruments>
<CsScore>
i 1 0 5
e
```

```
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*invalue*

## Credits

Author: Matt Ingalls

i-rate version new in Csound 6.04

## outx

`outx` — Writes 16-channel audio data to an external device or stream.

## Description

Writes 16-channel audio data to an external device or stream.

## Syntax

```
outx asig1, asig2, asig3, asig4, asig5, asig6, asig7, asig8, \  
      asig9, asig10, asig11, asig12, asig13, asig14, asig15, asig16
```

## Performance

`outx` outputs 16 channels of audio.

## See Also

*out32, outc, outch, outz*

## Credits

Author: John ffitc  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2000

New in Csound Version 4.07

# outz

outz — Writes multi-channel audio data from a ZAK array to an external device or stream.

## Description

Writes multi-channel audio data from a ZAK array to an external device or stream.

## Syntax

```
outz ksig1
```

## Performance

*outz* outputs from a ZAK array for *nchnls* of audio.

## See Also

*out32*, *outc*, *outch*, *outx*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2000

New in Csound Version 4.06

# p5gconnect

p5gconnect — Reads data from a P5 Glove controller.

## Description

Opens and at control-rate polls a P5 Glove controller.

## Syntax

```
p5gconnect
```

## Initialization

The opcode locates a P5 Glove attached to the computer by USB, and starts a listener thread to poll the device.

## Performance

Every control cycle the glove is polled for its position, and finger and button states. These values are read by the *p5gdata* opcode.

## Example

Here is an example of the p5g opcodes. It uses the file *p5g.csd* [examples/p5g.csd].

### Example 655. Example of the p5g opcodes.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
+rtaudio=alsa -o dac:hw:0
</CsOptions>
<CsInstruments>
nchnls = 1
ksmps = 1000

#define P5G_BUTTONS      #0#
#define P5G_BUTTON_A    #1#
#define P5G_BUTTON_B    #2#
#define P5G_BUTTON_C    #4#
#define P5G_JUSTPUSH    #8#
#define P5G_JUSTPU_A    #9#
#define P5G_JUSTPU_B    #10#
#define P5G_JUSTPU_C    #12#
#define P5G_RELEASED    #16#
#define P5G_RELSED_A    #17#
#define P5G_RELSED_B    #18#
#define P5G_RELSED_C    #20#
#define P5G_FINGER_INDEX #32#
#define P5G_FINGER_MIDDLE #33#
#define P5G_FINGER_RING  #34#
```

```

#define P5G_FINGER_PINKY #35#
#define P5G_FINGER_THUMB #36#
#define P5G_DELTA_X      #37#
#define P5G_DELTA_Y      #38#
#define P5G_DELTA_Z      #39#
#define P5G_DELTA_XR     #40#
#define P5G_DELTA_YR     #41#
#define P5G_DELTA_ZR     #42#
#define P5G_ANGLES      #43#

gka  init 0
gkp  init 0

instr 1
  p5gconnect
  ka  p5gdata $P5G_JUSTPU_A.
  kc  p5gdata $P5G_BUTTON_C.
  ; If the A button is just pressed then activate a note
  if (ka==0) goto ee
  event "i", 2, 0, 2

ee:
  gka p5gdata $P5G_DELTA_X.
  gkp p5gdata $P5G_DELTA_Y.
  printk2 gka
  printk2 gkp
  if (kc==0) goto ff
  printks "turning off (%d)\n", 0, kc
  turnoff

ff:
endin

instr 2
  a1 oscil ampdbfs(gkp), 440+100*gka, 1
  ;; a1 oscil 10000, 440, 1
  out a1
endin

</CsInstruments>

<CsScore>
f1 0 4096 10 1
i1 0 300

</CsScore>

</CsoundSynthesizer>

```

## See Also

*p5gdata*,

## Credits

Author: John ffitch  
 Codemist Ltd  
 2009

New in version 5.12

# p5gdata

p5gdata — Reads data fields from an external P5 Glove.

## Description

Reads data fields from a P5 Glove controller.

## Syntax

```
kres p5gdata kcontrol
```

## Initialization

This opcode must be used in conjunction with a running *p5gconnect* opcode.

## Performance

*kcontrol* -- the code for which control to read

On each access a particular data item of the P5 glove is read. The currently implemented controls are given below, together with the macro name defined in the file *p5g\_mac*:

0 (P5G\_BUTTONS): returns a bit pattern for all buttons that were pressed.

1 (P5G\_BUTTON\_A): returns 1 if the button has just been pressed, or 0 otherwise.

2 (P5G\_BUTTON\_B): as above.

4 (P5G\_BUTTON\_C): as above.

8 (P5G\_JUSTPUSH): returns a bit pattern for all buttons that have just been pressed.

9 (P5G\_JUSTPU\_A): returns 1 if the A button has just been pressed.

10 (P5G\_JUSTPU\_B): as above.

12 (P5G\_JUSTPU\_C): as above.

16 (P5G\_RELEASED): returns a bit pattern for all buttons that have just been released.

17 (P5G\_RELESED\_A): returns 1 if the A button has just been released.

18 (P5G\_RELESED\_B): as above.

20 (P5G\_RELESED\_C): as above.

32 (P5G\_FINGER\_INDEX): returns the clench value of the index finger.

33 (P5G\_FINGER\_MIDDLE): as above.

34 (P5G\_FINGER\_RING): as above.

35 (P5G\_FINGER\_PINKY): as above with little finger.

36 (P5G\_FINGER\_THUMB): as above.

37 (P5G\_DELTA\_X): The X position of the glove.

38 (P5G\_DELTA\_Y): The Y position of the glove.

39 (P5G\_DELTA\_Z): The Z position of the glove.

40 (P5G\_DELTA\_XR): The X axis change (angle).

41 (P5G\_DELTA\_YR): as above.

42 (P5G\_DELTA\_ZR): as above.

43 (P5G\_ANGLES): The general angle

## Examples

See the example for *p5gconnect*.

## See Also

*p5gconnect*,

## Credits

Author: John ffitc  
Codemist Ltd  
2009

New in version 5.12

# p

p — Show the value in a given p-field.

## Description

Show the value in a given p-field.

## Syntax

`p(x)`

This function works at i-rate and k-rate.

## Initialization

`x` -- the number of the p-field.

## Performance

The value returned by the *p* function is the value in a p-field.

## Examples

Here is an example of the p opcode. It uses the file *p.csd* [examples/p.csd].

### Example 656. Example of the p opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o p.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Get the value in the fourth p-field, p4.
i1 = p(4)

print i1
endin
```

```
</CsInstruments>
<CsScore>

; p4 = value to be printed.
; Play Instrument #1 for one second, p4 = 50.375.
i 1 0 1 50.375
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like:

```
instr 1: i1 = 50.375
```

## Credits

Example written by Kevin Conder.

# pan2

pan2 — Distribute an audio signal across two channels.

## Description

Distribute an audio signal across two channels with a choice of methods.

## Syntax

```
a1, a2 pan2 asig, xp [, imode]
```

## Initialization

*imode* (optional) -- mode of the stereo positioning algorithm. 0 signifies equal power (harmonic) panning, 1 means the square root method, 2 means simple linear and 3 means an alternative equal-power pan (based on an UDO). The default value is 0.

## Performance

*pan2* takes an input signal *asig* and distributes it across two outputs (essentially stereo speakers) according to the control *xp* which can be k- or a-rate. A zero value for *xp* indicates hard left, and a 1 is hard right.

## Example

Here is an example of the pan2 opcodes. It uses the file *pan2.csd* [examples/pan2.csd].

### Example 657. Example of the pan2 opcodes.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if real audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pan2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine ftgen 0, 0, 2^10, 10, 1

instr 1

kline line 0, p3, 1      ; straight line
ain oscili .6, 440, giSine ; audio signal..
aL,aR pan2 ain, kline   ; sent across image
```

```
outs aL, aR  
  
endin  
</CsInstruments>  
<CsScore>  
i1 0 5  
e  
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Author: John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK  
September 2007

New in version 5.07

# pan

pan — Distribute an audio signal amongst four channels.

## Description

Distribute an audio signal amongst four channels with localization control.

## Syntax

```
a1, a2, a3, a4 pan asig, kx, ky, ifn [, imode] [, ioffset]
```

## Initialization

*ifn* -- function table number of a stored pattern describing the amplitude growth in a speaker channel as sound moves towards it from an adjacent speaker. Requires extended guard-point.

*imode* (optional) -- mode of the *kx*, *ky* position values. 0 signifies raw index mode, 1 means the inputs are normalized (0 - 1). The default value is 0.

*ioffset* (optional) -- offset indicator for *kx*, *ky*. 0 infers the origin to be at channel 3 (left rear); 1 requests an axis shift to the quadraphonic center. The default value is 0.

## Performance

*pan* takes an input signal *asig* and distributes it amongst four outputs (essentially quad speakers) according to the controls *kx* and *ky*. For normalized input (mode=1) and no offset, the four output locations are in order: left-front at (0,1), right-front at (1,1), left-rear at the origin (0,0), and right-rear at (1,0). In the notation (*kx*, *ky*), the coordinates *kx* and *ky*, each ranging 0 - 1, thus control the 'rightness' and 'forwardness' of a sound location.

Movement between speakers is by amplitude variation, controlled by the stored function table *ifn*. As *kx* goes from 0 to 1, the strength of the right-hand signals will grow from the left-most table value to the right-most, while that of the left-hand signals will progress from the right-most table value to the left-most. For a simple linear pan, the table might contain the linear function 0 - 1. A more correct pan that maintains constant power would be obtained by storing the first quadrant of a sinusoid. Since *pan* will scale and truncate *kx* and *ky* in simple table lookup, a medium-large table (say 8193) should be used.

*kx*, *ky* values are not restricted to 0 - 1. A circular motion passing through all four speakers (inscribed) would have a diameter of root 2, and might be defined by a circle of radius  $R = \text{root } 1/2$  with center at (.5,.5). *kx*, *ky* would then come from  $R\cos(\text{angle})$ ,  $R\sin(\text{angle})$ , with an implicit origin at (.5,.5) (i.e. *ioffset* = 1). Unscaled raw values operate similarly. Sounds can thus be located anywhere in the polar or Cartesian plane; points lying outside the speaker square are projected correctly onto the square's perimeter as for a listener at the center.

## Example

Here is an example of the pan opcodes. It uses the file *pan.csd* [examples/pan.csd].

### Example 658. Example of the pan opcodes.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if real audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pan.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 4
0dbfs = 1

instr 1

kcps = p4
k1 phasor kcps ; "fraction" of circle - controls speed of rotation - can be negative
k2 tablei k1, 1, 1 ; sin of angle (sinusoid in f1)
k3 tablei k1, 1, 1, .25, 1 ; cos of angle (sin offset 1/4 circle)
arnd randomi 400, 1000, 50 ; produce random values
asig poscil .7, arnd, 1 ; audio signal..

a1,a2,a3,a4 pan asig, k2/2, k3/2, 2, 1, 1 ; sent in a circle (f2=1st quad sin)
outq a1, a2, a3, a4

endin
</CsInstruments>
<CsScore>

f1 0 8192 10 1
f2 0 8193 9 .25 1 0

i1 0 10 .2 ;move to the tight
i1 11 10 -.2 ;move to the left
e

</CsScore>
</CsoundSynthesizer>
```

## pareq

pareq — Implementation of Zoelzer's parametric equalizer filters.

### Description

Implementation of Zoelzer's parametric equalizer filters, with some modifications by the author.

The formula for the low shelf filter is:

$$\begin{aligned}\omega &= 2\pi f/sr \\ K &= \tan(\omega/2) \\ b0 &= 1 + \sqrt{2V}K + V K^2 \\ b1 &= 2(V K^2 - 1) \\ b2 &= 1 - \sqrt{2V}K + V K^2 \\ a0 &= 1 + K/Q + K^2 \\ a1 &= 2(K^2 - 1) \\ a2 &= 1 - K/Q + K^2\end{aligned}$$

The formula for the high shelf filter is:

$$\begin{aligned}\omega &= 2\pi f/sr \\ K &= \tan((\pi - \omega)/2) \\ b0 &= 1 + \sqrt{2V}K + V K^2 \\ b1 &= -2(V K^2 - 1) \\ b2 &= 1 - \sqrt{2V}K + V K^2 \\ a0 &= 1 + K/Q + K^2 \\ a1 &= -2(K^2 - 1) \\ a2 &= 1 - K/Q + K^2\end{aligned}$$

The formula for the peaking filter is:

$$\begin{aligned}\omega &= 2\pi f/sr \\ K &= \tan(\omega/2) \\ b0 &= 1 + V K/2 + K^2 \\ b1 &= 2(K^2 - 1) \\ b2 &= 1 - V K/2 + K^2 \\ a0 &= 1 + K/Q + K^2 \\ a1 &= 2(K^2 - 1) \\ a2 &= 1 - K/Q + K^2\end{aligned}$$

## Syntax

```
ares pareq asig, kc, kv, kq [, imode] [, iskip]
```

## Initialization

*imode* (optional, default: 0) -- operating mode

- 0 = Peaking
- 1 = Low Shelving
- 2 = High Shelving

*iskip* (optional, default=0) -- if non zero skip the initialisation of the filter. (New in Csound version 4.23f13 and 5.0)

## Performance

*kc* -- center frequency in peaking mode, corner frequency in shelving mode.

*kv* -- amount of boost or cut. A value less than 1 is a cut. A value greater than 1 is a boost. A value of 1 is a flat response.

*kq* -- Q of the filter (sqrt(.5) is no resonance)

*asig* -- the incoming signal

## Examples

Here is an example of the `pareq` opcode. It uses the file `pareq.csd` [examples/pareq.csd].

### Example 659. Example of the `pareq` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc      -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o pareq.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

instr 15
  ifc      =      p4          ; Center / Shelf
  kq       =      p5          ; Quality factor sqrt(.5) is no resonance
  kv       =      ampdb(p6)   ; Volume Boost/Cut
  imode    =      p7          ; Mode 0=Peaking EQ, 1=Low Shelf, 2=High Shelf
```

```
kfc      linseg  ifc*2, p3, ifc/2
asig     rand    5000                ; Random number source for testing
aout     pareq   asig, kfc, kv, kq, imode ; Parmetric equalization
outs     outs    aout, aout           ; Output the results
endin

</CsInstruments>
<CsScore>

; SCORE:
;  Sta  Dur  Fcenter  Q          Boost/Cut(dB)  Mode
i15 0    1    10000   .2          12             1
i15 +    .    5000    .2          12             1
i15 .    .    1000    .707       -12            2
i15 .    .    5000    .1         -12            0
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Hans Mikelson  
December 1998

New in Csound version 3.50

# partials

partials — Partial track spectral analysis.

## Description

The `partials` opcode takes two input PV streaming signals containing `AMP_FREQ` and `AMP_PHASE` signals (as generated for instance by `pvsifd` or in the first case, by `pvsanal`) and performs partial track analysis, as described in Lazzarini et al, "Time-stretching using the Instantaneous Frequency Distribution and Partial Tracking", Proc.of ICMC05, Barcelona. It generates a `TRACKS` PV streaming signal, containing amplitude, frequency, phase and track ID for each output track. This type of signal will contain a variable number of output tracks, up to the total number of analysis bins contained in the inputs ( $\text{fftsize}/2 + 1$  bins). The second input (`AMP_PHASE`) is optional, as it can take the same signal as the first input. In this case, however, all phase information will be `NULL` and resynthesis using phase information cannot be performed.

## Syntax

```
ftrks partials ffr, fphs, kthresh, kminpts, kmaxgap, imaxtracks
```

## Performance

*ftrks* -- output pv stream in `TRACKS` format

*ffr* -- input pv stream in `AMP_FREQ` format

*fphs* -- input pv stream in `AMP_PHASE` format

*kthresh* -- analysis threshold factor, defined between -1 and 1. If non-negative, the analysis threshold is taken to be relative to the maximum magnitude in each analysis frame ( $\text{kthresh} * \text{max\_magnitude}$ ). If negative, the absolute threshold value is relative to `0dbfs` ( $\text{kthresh} * \text{0dbfs}$ ). Tracks below this threshold will be discarded.

*kminpoints* -- minimum number of time points for a detected peak to make a track (1 is the minimum). Since this opcode works with streaming signals, larger numbers will increase the delay between input and output, as we have to wait for the required minimum number of points.

*kmaxgap* -- maximum gap between time-points for track continuation ( $> 0$ ). Tracks that have no continuation after `kmaxgap` will be discarded.

*imaxtracks* -- maximum number of analysis tracks (number of bins  $\geq$  `imaxtracks`)

## Example

Here is an example of the `partials` opcodes. It uses the file `partials.csd` [examples/partial.csd].

### Example 660. Example of the `partials` opcodes.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if real audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o partials.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ain  diskin2 "fox.wav", 1
fs1,fsi2 pvsifd ain,2048,512,1 ; ifd analysis
fst  partials fs1,fsi2,.003,1,3,500 ; partial tracking
aout resyn fst, 1, 1.5, 500, 1 ; resynthesis (up a 5th)
      outs aout, aout

endin
</CsInstruments>
<CsScore>
f 1 0 4096 10 1

i 1 0 2.8
e
</CsScore>
</CsoundSynthesizer>
```

The example above shows partial tracking of an ifd-analysis signal and cubic-phase additive resynthesis with pitch shifting.

## Credits

Author: Victor Lazzarini  
June 2005

New plugin in version 5

November 2004.

# partikkel

*partikkel* — Granular synthesizer with "per grain" control over many of its parameters. Has a sync input to synchronize its internal grain scheduler clock to an external clock source.

## Description

*partikkel* was conceived after reading Curtis Roads' book "Microsound", and the goal was to create an opcode that was capable of all time-domain varieties of granular synthesis described in this book. The idea being that most of the techniques only differ in parameter values, and by having a single opcode that can do all varieties of granular synthesis makes it possible to interpolate between techniques. Granular synthesis is sometimes dubbed particle synthesis, and it was thought apt to name the opcode *partikkel* to distinguish it from other granular opcodes.

Some of the input parameters to *partikkel* is table numbers, pointing to tables where values for the "per grain" parameter changes are stored. *partikkel* can use single-cycle or complex (e.g. sampled sound) waveforms as source waveforms for grains. Each grain consists of a mix of 4 source waveforms. Individual tuning of the base frequency can be done for each of the 4 source waveforms. Frequency modulation inside each grain is enabled via an auxillary audio input (*awavfm*). Trainlet synthesis is available, and trainlets can be mixed with wavetable based grains. Up to 8 separate audio outputs can be used. Fractional output values will distribute a grain between two outputs and a "panning law" table can be devised to control the amplitude scaling between output pairs.

## Syntax

```
a1 [, a2, a3, a4, a5, a6, a7, a8] partikkel agrainfreq, \  
    kdistribution, idisttab, async, kenv2amt, ienv2tab, ienv_attack, \  
    ienv_decay, ksustain_amount, ka_d_ratio, kduration, kamp, igainmasks, \  
    kwavfreq, ksweepshape, iwavfreqstarttab, iwavfreqendtab, awavfm, \  
    ifmampstab, kfmenv, icosine, ktraincps, knumpartials, kchroma, \  
    ichannelmasks, krandommask, kwaveform1, kwaveform2, kwaveform3, \  
    kwaveform4, iwaveampstab, asamplepos1, asamplepos2, asamplepos3, \  
    asamplepos4, kwavekey1, kwavekey2, kwavekey3, kwavekey4, imax_grains \  
    [, iopcode_id, ipanlaws]
```

## Initialization

*idisttab* -- function table number, distribution for random grain displacements over time. The table values are interpreted as "displacement amount" scaled by 1/grainrate. This means that a value of 0.5 in the table will displace a grain by half the grainrate period. The table values are read randomly, and scaled by *kdistribution*. For realistic stochastic results, it is advisable not to use a too small table size, as this limits the amount of possible displacement values. This can also be utilized for other purposes, e.g. using quantized displacement values to work with controlled time displacement from the periodic grain rate. If *kdistribution* is negative, the table values will be read sequentially. A default table might be selected by using -1 as the ftable number, for *idisttab* the default uses a zero distribution (no displacement).

*ienv\_attack* -- function table number, attack shape of grain. Needs extended guard point. A default table might be selected by using -1 as the ftable number, for *ienv\_attack* the default uses a square window (no enveloping).

*ienv\_decay* -- function table number, decay shape of grain. Needs extended guard point. A default table might be selected by using -1 as the ftable number, for *ienv\_decay* the default uses a square window (no enveloping).

*ienv2tab* -- function table number, additional envelope applied to grain, done after attack and decay envelopes. Can be used e.g. for fof formant synthesis. Needs extended guard point. A default table might be selected by using -1 as the ftable number, for *ienv2tab* the default uses a square window (no enveloping).

*icosine* -- function table number, must contain a cosine, used for trainlets. Table size should be at least 2048 for good quality trainlets.

*igainmasks* -- function table number, gain per grain. The sequence of values in the table is as follows: index 0 is used as a loop start point in reading the values, index 1 is used as a loop end point. Remaining indices contain gain values (normally in range 0 - 1, but other values are allowed, negative values will invert phase of waveform inside grain) for a sequence of grains, these are read at grain rate enabling exact patterns of "gain per grain". The loop start and end points are zero based with an origin at index 2, e.g. a loop start value of 0 and loop end value of 3 will read indices 2,3,4,5 in a loop at grain rate. A default table might be selected by using -1 as the ftable number, for *igainmasks* the default disables gain masking (all grains are given a gain masking value of 1).

*ichannelmasks* -- function table number, see *igainmasks* for a description of how the values in the table are read. Range is 0 to N, where N is the number of output channels. A value of zero will send the grain to audio output 1 from the opcode. Fractional values are allowed, e.g. a value of 3.5 will mix the grain equally to outputs 4 and 5. The channelmask value wraps around from the last to the first output, so that a value of N-0.5 will mix the grain equally between the last and the first output. If another panning law between outputs is desired, this can be described in the *ipanlaws* table. The user is responsible for keeping the values in range, the opcode will crash with out of range values. A default table might be selected by using -1 as the ftable number, for *ichannelmasks* the default disables channel masking (all grains are given a channel masking value of 0 and are sent to *partikkel* audio out 1).

*iwavfreqstarttab* -- function table number, see *igainmasks* for a description of how the values in the table are read. Start frequency multiplier for each grain. Pitch will glide from start frequency to end frequency following a line or curve as set by *ksweepshape*. A default table might be selected by using -1 as the ftable number, for *iwavfreqstarttab* the default uses a multiplier of 1, disabling any start frequency modification.

*iwavfreqendtab* -- function table number, see *iwavfreqstarttab*. End frequency multiplier for each grain. A default table might be selected by using -1 as the ftable number, for *iwavfreqendtab* the default uses a multiplier of 1, disabling any end frequency modification.

*ifmamptab* -- function table number, see *igainmasks* for a description of how the values in the table are read. Frequency modulation index per grain. The signal *awavfm* will be multiplied by values read from this table. A default table might be selected by using -1 as the ftable number, for *ifmamptab* the default uses 1 as the index multiplier, enabling fm for all grains.

*iwaveamptab* -- function table number, the indices are read in a similar way to what is used for *igainmasks*. Index 0 is used as a loop start point, and index 1 is used as a loop end point. The rest of the indices are read in groups of 5, where each value represent a gain value for each of the 4 source waveforms, and the 5th value represent trainlet amplitude. A default table might be selected by using -1 as the ftable number, for *iwaveamptab* the default uses an equal mix of all 4 source waveforms (each with an amplitude of 0.5) and setting trainlet amp to zero.

Computation of trainlets can be CPU intensive, and setting *ktrainamp* to zero will skip most of the trainlet computations. Trainlets will be normalized to peak (*ktrainamp*), compensating for amplitude variations caused by variations in *kpartials* and *kchroma*.

*imax\_grains* -- maximum number of grains per k-period. Estimating a large value should not affect performance, exceeding this value will lead to "oldest grains" being deleted.

*iopcode\_id* -- the opcode id, linking an instance of *partikkel* to an instance of *partikkelsync*, the linked *partikkelsync* will output trigger pulses synchronized to *partikkel*'s grain maker scheduler. The default value is zero, which means no connection to any *partikkelsync* instances.

*ipanlaws* -- function table number. The table describes the panning curve used for fractional channelmask values. Fractional channelmask values will mix a grain to two neighbouring outputs, with the relative gain set by the fractional value. By default (if no *ipanlaws* table is described, a linear gain relationship is used, so that a channelmask value of e.g. 1.5 distributes the grain with 0.5 gain to output 2 and 0.5 gain to output 3. The *ipanlaws* table can be used to describe other gain control curves (panning laws). The table should contain 8 such gain control curves, each governing the panning between two neighbouring outputs. The curves should appear one after another in the table, in a concatenated fashion. GEN 18 can be used to create this table from separate panning curve tables (see example below). The first curve describes the panning law between output 1 and output 2, the next is for panning between outputs 2 and 3, and so on. The last curve describes the panning law between the last and the first output. The table is indexed by the channelmask value such that one output (of an output pair governed by the panning law) uses the index ( $\text{tablesize}/8 * \text{channelmask}$ ) while the other of the two outputs reads the value at index ( $\text{tablesize}/8 * (\text{int}(\text{channelmask} + 1) - \text{frac}(\text{channelmask}))$ ). This means that if the panning law value halfway between these two channel masks is e.g. 0.7 (which would give approximately equal power panning), then each of those two outputs will use 0.7 as the gain value.

## Performance

*xgrainfreq* -- number of grains per second. A value of zero is allowed, and this will defer all grain scheduling to the sync input.

*async* -- sync input. Input values are added to the phase value of the internal grain maker clock, enabling tempo synchronization with an external clock source. As this is an a-rate signal, inputs are usually pulses of length  $1/sr$ . Using such pulses, the internal phase value can be "nudged" up or down, enabling soft or hard synchronization. Negative input values decrements the internal phase, while positive values in the range 0 to 1 increments the internal phase. An input value of 1 will always make *partikkel* generate a grain. If the value remains at 1, the internal grain scheduler clock will pause but any currently playing grains will still play to end.

*kdistribution* -- periodic or stochastic distribution of grains, 0 = periodic. Stochastic grain displacement is in the range of  $\text{kdistribution}/\text{grainrate}$  seconds. The stochastic distribution profile (random distribution) can be set in the *idistab* table. If *kdistribution* is negative, the result is deterministic time displacement as described by *idistab* (sequential read of displacement values). Maximum grain displacement in all cases is limited to 10 seconds, and a grain will keep the values (duration, pitch etc) it was given when it was first generated (before time displacement). Since grain displacement is relative to the grain rate, displacement amount is undefined at 0Hz grain rate and *kdistribution* is completely disabled in this case.

*kenv2amt* -- amount of enveloping for the secondary envelope for each grain. Range 0 to 1, where 0 is no secondary enveloping (square window), a value of 0.5 will use an interpolation between a square window and the shape set by *ienv2tab*.

*ksustain\_amount* -- sustain time as fraction of grain duration. I.e. balance between enveloped time(attack+decay) and sustain level time. The sustain level is taken from the last value of the *ienv\_attack* ftable.

*ka\_d\_ratio* -- balance between attack time and decay time. For example, with *ksustain\_amount* set to 0.5 and *ka\_d\_ratio* set to 0.5, the attack envelope of each grain will take 25% of the grain duration, full amplitude (sustain) will be held for 50% of the grain duration, and the decay envelope will take the remaining 25% of the grain duration.

*kduration* -- grain duration in milliseconds.

*kamp* -- amplitude scaling of the opcode's output. Multiplied by per grain amplitude read from *igainmasks*. Source waveform playback inside grains can consume a significant amount of CPU cycles, especially if grain duration is long so that we have a lot of overlapping grains. Setting *kamp* to zero will skip waveform playback inside grains (and not generate any sound, obviously). This can be used as a "soft" bypass method if we want to keep the opcode active but silent for some periods of time.

*kwavfreq* -- transposition scaling. Multiplied with start and end transposition values read from *iwavfreqs-tarttab* and *iwavfrendtab*.

*ksweepshape* -- transposition sweep shape, controls the curvature of the transposition sweep. Range 0 to 1. Low values will hold the transposition at the start value longer and then drop to the end value quickly, high values will drop to the end value quickly. A value of 0.5 will give a linear sweep. A value of exactly 0 will bypass sweep and only use the start frequency, while a value of exactly 1 will bypass sweep and only use the end frequency. The sweep generator might be slightly inaccurate in hitting the end frequency when using a steep curve and very long grains.

*awavfm* -- audio input for frequency modulation inside grain.

*kfmenv* -- function table number, envelope for FM modulator signal enabling the modulation index to change over the duration of a grain.

*ktraincps* -- trainlet fundamental frequency.

*knumpartials* -- number of partials in trainlets.

*kchroma* -- chroma of trainlets. A value of 1 give equal amplitude to each partial, higher values will reduce the amplitude of lower partials while strengthening the amplitude of the higher partials.

*krandommask* -- random masking (muting) of individual grains. Range 0 to 1, where a value of 0 will give no masking (all grains are played), and a value of 1 will mute all grains.

*kwaveform1* -- table number for source waveform 1.

*kwaveform2* -- table number for source waveform 2.

*kwaveform3* -- table number for source waveform 3.

*kwaveform4* -- table number for source waveform 4.

*asamplepos1* -- start position for reading source waveform 1 (in range 0..1).

*asamplepos2* -- start position for reading source waveform 2.

*asamplepos3* -- start position for reading source waveform 3.

*asamplepos4* -- start position for reading source waveform 4.

*kwavekey1* -- original key of source waveform 1. Can be used to transpose each source waveform independently.

*kwavekey2* -- as *kwavekey1*, but for source waveform 2.

*kwavekey3* -- as *kwavekey1*, but for source waveform 3.

*kwavekey4* -- as *kwavekey1*, but for source waveform 4.

## Examples

Here is an example of the *partikkel* opcode. It uses the file *partikkel.csd* [examples/partikkel.csd].

**Example 661. Example of the partikkel opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if real audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o partikkel.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 20
nchnls = 2

giSine ftgen 0, 0, 65537, 10, 1
giCosine ftgen 0, 0, 8193, 9, 1, 1, 90

instr 1

kgrainfreq = 200 ; 4 grains per second
kdistribution = 0 ; periodic grain distribution
idisttab = -1 ; (default) flat distribution used for grain distribution
async = 0 ; no sync input
kenv2amt = 0 ; no secondary enveloping
ienv2tab = -1 ; default secondary envelope (flat)
ienv_attack = -1 ; ; default attack envelope (flat)
ienv_decay = -1 ; ; default decay envelope (flat)
ksustain_amount = 0.5 ; time (in fraction of grain dur) at sustain level for each grain
ka_d_ratio = 0.5 ; balance between attack and decay time
kduration = (0.5/kgrainfreq)*1000 ; set grain duration relative to grain rate
kamp = 5000 ; amp
igainmasks = -1 ; (default) no gain masking
kwavfreq = 440 ; fundamental frequency of source waveform
ksweepshape = 0 ; shape of frequency sweep (0=no sweep)
iwavfreqstarttab = -1 ; default frequency sweep start (value in table = 1, which give no frequency modification)
iwavfreqendtab = -1 ; default frequency sweep end (value in table = 1, which give no frequency modification)
awavfm = 0 ; no FM input
ifmampmtab = -1 ; default FM scaling (=1)
kfmenv = -1 ; default FM envelope (flat)
icosine = giCosine ; cosine ftable
kTrainCps = kgrainfreq ; set trainlet cps equal to grain rate for single-cycle trainlet in each grain
knumpartials = 3 ; number of partials in trainlet
kchroma = 1 ; balance of partials in trainlet
ichannelmasks = -1 ; (default) no channel masking, all grains to output 1
krandommask = 0 ; no random grain masking
kwaveform1 = giSine ; source waveforms
kwaveform2 = giSine ;
kwaveform3 = giSine ;
kwaveform4 = giSine ;
iwaveampmtab = -1 ; (default) equal mix of all 4 source waveforms and no amp for trainlets
asamplepos1 = 0 ; phase offset for reading source waveform
asamplepos2 = 0 ;
asamplepos3 = 0 ;
asamplepos4 = 0 ;
kwavekey1 = 1 ; original key for source waveform
kwavekey2 = 1 ;
kwavekey3 = 1 ;
kwavekey4 = 1 ;
imax_grains = 100 ; max grains per k period

asig partikkel kgrainfreq, kdistribution, idisttab, async, kenv2amt, ienv2tab, \

```

```

ienv_attack, ienv_decay, ksustain_amount, ka_d_ratio, kduration, kamp, igainmasks, \
kwavfreq, ksweepshape, iwavfreqstarttab, iwavfreqendtab, awavfm, \
ifmamptab, kfmenv, icosine, kTrainCps, knumpartials, \
kchroma, ichannelmasks, krandommask, kwaveform1, kwaveform2, kwaveform3, kwaveform4, \
iwaveamptab, asamplepos1, asamplepos2, asamplepos3, asamplepos4, \
kwavekey1, kwavekey2, kwavekey3, kwavekey4, imax_grains

outs asig, asig
endin

</CsInstruments>
<CsScore>
i1 0 5 ; partikkel
e
</CsScore>
</CsoundSynthesizer>

```

Here is another example of the partikkel opcode. It uses the file *partikkel-2.csd* [examples/partikkel-2.csd].

### Example 662. Example 2 of the partikkel opcode.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out
-odac          ;;RT audio
; For Non-realtime ouput leave only the line below:
; -o partikkel.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 20
nchnls = 2

; Example by Joachim Heintz and Oeyvind Brandtsegg 2008

giCosine ftgen 0, 0, 8193, 9, 1, 1, 90 ; cosine
giDisttab ftgen 0, 0, 32768, 7, 0, 32768, 1 ; for kdistribution
giFile ftgen 0, 0, 0, 1, "fox.wav", 0, 0, 0 ; soundfile for source waveform
giWin ftgen 0, 0, 4096, 20, 9, 1 ; grain envelope
giPan ftgen 0, 0, 32768, -21, 1 ; for panning (random values between 0 and 1)

; *****
; partikkel example, processing of soundfile
; uses the file "fox.wav"
; *****
instr 1

/*score parameters*/
ispeed = p4 ; 1 = original speed
igrainrate = p5 ; grain rate
igrainsize = p6 ; grain size in ms
icent = p7 ; transposition in cent
iposrand = p8 ; time position randomness (offset) of the pointer in ms
icentrand = p9 ; transposition randomness in cents
ipan = p10 ; panning narrow (0) to wide (1)
idist = p11 ; grain distribution (0=periodic, 1=scattered)

/*get length of source wave file, needed for both transposition and time pointer*/
ifilen tablen giFile
ifildur = ifilen / sr

/*sync input (disabled)*/

```

```

async      = 0

/*grain envelope*/
kenv2amt   = 1 ; use only secondary envelope
ienv2tab   = giWin ; grain (secondary) envelope
ienv_attack = -1 ; default attack envelope (flat)
ienv_decay = -1 ; default decay envelope (flat)
ksustain_amount = 0.5 ; no meaning in this case (use only secondary envelope, ienv2tab)
ka_d_ratio = 0.5 ; no meaning in this case (use only secondary envelope, ienv2tab)

/*amplitude*/
kamp       = 0.4*0dbfs ; grain amplitude
igainmask  = -1 ; (default) no gain masking

/*transposition*/
kcentrand  rand icentrand ; random transposition
iorig      = 1 / ifildur ; original pitch
kwavfreq   = iorig * cent(icent + kcentrand)

/*other pitch related (disabled)*/
ksweepshape = 0 ; no frequency sweep
iwavfreqstarttab = -1 ; default frequency sweep start
iwavfreqendtab = -1 ; default frequency sweep end
awavfm      = 0 ; no FM input
ifmamptab   = -1 ; default FM scaling (=1)
kfmenv      = -1 ; default FM envelope (flat)

/*trainlet related (disabled)*/
icosine     = giCosine ; cosine ftable
kTrainCps   = igrainrate ; set trainlet cps equal to grain rate for single-cycle trainlet in each grain
knumpartials = 1 ; number of partials in trainlet
kchroma     = 1 ; balance of partials in trainlet

/*panning, using channel masks*/
imid        = .5; center
ileftmost   = imid - ipan/2
irightmost  = imid + ipan/2
giPanthis   ftgen 0, 0, 32768, -24, giPan, ileftmost, irightmost ; rescales giPan according to ipan
    tableiw 0, 0, giPanthis ; change index 0 ...
    tableiw 32766, 1, giPanthis ; ... and 1 for ichannelmasks
ichannelmasks = giPanthis ; ftable for panning

/*random gain masking (disabled)*/
krandommask = 0

/*source waveforms*/
kwaveform1  = giFile ; source waveform
kwaveform2  = giFile ; all 4 sources are the same
kwaveform3  = giFile
kwaveform4  = giFile
iwaveamptab = -1 ; (default) equal mix of source waveforms and no amplitude for trainlets

/*time pointer*/
afilposphas phasor ispeed / ifildur
/*generate random deviation of the time pointer*/
iposrandsec = iposrand / 1000 ; ms -> sec
iposrand    = iposrandsec / ifildur ; phase values (0-1)
krndpos     linrand iposrand ; random offset in phase values
/*add random deviation to the time pointer*/
asamplepos1 = afileposphas + krndpos; resulting phase values (0-1)
asamplepos2 = asamplepos1
asamplepos3 = asamplepos1
asamplepos4 = asamplepos1

/*original key for each source waveform*/
kwavekey1   = 1
kwavekey2   = kwavekey1

```

```

kwavekey3 = kwavekey1
kwavekey4 = kwavekey1

/* maximum number of grains per k-period*/
imax_grains = 100

aL, aR partikkel igrainrate, idist, giDisttab, async, kenv2amt, ienv2tab, \
ienv_attack, ienv_decay, ksustain_amount, ka_d_ratio, igrainsize, kamp, igainmasks, \
kwavfreq, ksweepshape, iwavfreqstarttab, iwavfreqendtab, awavfm, \
ifmamptab, kfmenv, icosine, kTrainCps, knumpartials, \
kchroma, ichannelmasks, krandommask, kwaveform1, kwaveform2, kwaveform3, kwaveform4, \
iwaveamptab, asamplepos1, asamplepos2, asamplepos3, asamplepos4, \
kwavekey1, kwavekey2, kwavekey3, kwavekey4, imax_grains

outs aL, aR

endin

</CsInstruments>
<CsScore>
;il st dur speed grate gsize cent posrnd cntrnd pan dist
i1 0 2.757 1 200 15 0 0 0 0 0
s
i1 0 2.757 1 200 15 400 0 0 0 0
s
i1 0 2.757 1 15 450 400 0 0 0 0
s
i1 0 2.757 1 15 450 400 0 0 0 0.4
s
i1 0 2.757 1 200 15 0 400 0 0 1
s
i1 0 5.514 .5 200 20 0 0 600 .5 1
s
i1 0 11.028 .25 200 15 0 1000 400 1 1

</CsScore>

</CsoundSynthesizer>

```

Here is an example of using panning laws with channelmasks in partikkel. It uses the file *partikkel-panlaws.csd* [examples/partikkel-panlaws.csd].

### Example 663. Example with panning laws with channel masks.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out
-odac          ;;RT audio
; For Non-realtime ouput leave only the line below:
; -o partikkel-panlaws.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

nchnls = 4
0dbfs = 1

giSine ftgen 0, 0, 65536, 10, 1
giCosine ftgen 0, 0, 8192, 9, 1, 1, 90
giSigmoidRise ftgen 0, 0, 8193, 19, 0.5, 1, 270, 1
giSigmoidFall ftgen 0, 0, 8193, 19, 0.5, 1, 90, 1
giLinUp ftgen 0, 0, 8192, 7, 0, 8192, 1
giConcaveUp ftgen 0, 0, 8192, 16, 0, 8192, 1.5, 1

```

```
giConvexUp ftgen 0, 0, 8192, 16, 0, 8192, -1.5, 1
giPanLaws ftgen 0, 0, 8192, 18,
    giLinUp, 1, 0, 1023,
    giConcaveUp, 1, 1024, 2047,
    giConvexUp, 1, 2048, 3071,
    giSigmoRise, 1, 3072, 4095

instr 1

; channel masking table, using just one single mask here
ichannelmasks ftgentmp 0, 0, 32, -2, 0, 0, 0

; continuously write to masking table,
; slowly panning the grains from output to output
; over a 10 second period
kchn phasor 0.1
kchn = kchn*4
tablew kchn, 2, ichannelmasks

; init unused arate signals
awavfm = 0
asamplepos1 = 0
async = 0

a1,a2,a3,a4 partikkel 4, 0, -1, async, 0, -1, giSigmoRise, giSigmoFall,
    0.9, 0.5, 100, ampdbfs(-9), -1, 1, 0, -1, -1, awavfm,
    -1, -1, giCosine, 1, 1, 1, ichannelmasks, 0,
    giSine, giSine, giSine, giSine,
    -1, asamplepos1, asamplepos1, asamplepos1, asamplepos1,
    440, 440, 440, 440, 100, 1, giPanLaws

outch 1, a1, 2, a2, 3, a3, 4, a4

endin

</CsInstruments>
<CsScore>
i1 0 12
</CsScore>
</CsoundSynthesizer>
```

## See Also

*fof, fof2, fog, grain, grain2, grain3, granule, sndwarp, sndwarpst, syncgrain, syncloop, partikkelget partikkelset partikkelsync*

## Credits

Author: Thom Johansen  
Author: Torgeir Strand Henriksen  
Author: Øyvind Brandtsegg  
April 2007

Examples written by Joachim Heintz and Øyvind Brandtsegg.

New in version 5.06

# partikkelget

partikkelget — Get mask index for a specific mask parameter of a running *partikkel* instance.

## Description

*partikkelget* is an opcode for outputting *partikkel* mask index for a specific parameter. Used together with *partikkelset*, it can be used to synchronize partikkel masking between several running instances of the *partikkel* opcode. It can also be used to control other processes based on the internal mask index, for example to create more complex masking patterns than is available with the regular grain masking system.

## Syntax

```
kindex partikkelget kparameterindex, iopcode_id
```

## Initialization

*iopcode\_id* -- the opcode id, linking an instance of *partikkel* to an instance of *partikkelsync*.

## Performance

*kmaskindex* -- mask index output. Outputs the current mask index for the parameter specified with *kparameterindex* in the *partikkel* instance identified with *iopcode\_id*.

*kparameterindex* -- mask parameter. Selection of the masking parameter for which to output the current mask index. The different parameters are identified as:

- 0: gain mask
- 1: pitch sweep start mask
- 2: pitch sweep end mask
- 3: fm modulation index mask
- 4: channel mask
- 5: waveform mix mask

## Example

Here is an example of the *partikkelget* and *partikkelset* opcodes. It uses the file *partikkelgetset.csd* [examples/partikkelgetset.csd].

**Example 664. Example of manipulating the internal mask index of *partikkel*, based on the value of other mask indices in the same *partikkel* instance.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out
-odac          ;;RT audio
; For Non-realtime ouput leave only the line below:
-o partikkel-getset.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

nchnls = 2
Odbfs = 1

giSine ftgen 0, 0, 65536, 10, 1
giCosine ftgen 0, 0, 8192, 9, 1, 1, 90
giSigmoidRise ftgen 0, 0, 8193, 19, 0.5, 1, 270, 1
giSigmoidFall ftgen 0, 0, 8193, 19, 0.5, 1, 90, 1

instr 1

; freqstart masking table, simple ascending semitones pattern
iwavfreqstarttab ftgentmp 0, 0, 32, -2, 0, 11,
    semitone(0), semitone(1), semitone(2), semitone(3),
    semitone(4), semitone(5), semitone(6), semitone(7),
    semitone(8), semitone(9), semitone(10), semitone(11)

iwavfreqendtab ftgentmp 0, 0, 32, -2, 0, 11,
    semitone(0), semitone(1), semitone(2), semitone(3),
    semitone(4), semitone(5), semitone(6), semitone(7),
    semitone(8), semitone(9), semitone(10), semitone(11)

; channel masking table, simple panning back and forth
ichannelmasks ftgentmp 0, 0, 32, -2, 0, 7, 0.0, 0.25, 0.5, 0.75, 1.0, 0.75, 0.5, 0.25

; init unused arate signals
awavfm = 0
asamplepos1 = 0
async = 0

; system
id = 1

a1,a2 partikkel 6, 0, -1, async, 0, -1, giSigmoidRise, giSigmoidFall,
    0.9, 0.5, 100, ampdbfs(-9), -1, 1, 0, iwavfreqstarttab, iwavfreqendtab, awavfm,
    -1, -1, giCosine, 1, 1, 1, ichannelmasks, 0,
    giSine, giSine, giSine, giSine,
    -1, asamplepos1, asamplepos1, asamplepos1, asamplepos1,
    440, 440, 440, 440, 100, id

outch 1, a1, 2, a2

; using partikkelget and partikkelset to make the channelmask and freqmask indices
; interact to create a more complex pattern
kfqmask partikkelget 1, id
kchmask partikkelget 4, id
kcount1 init 0
kcount2 init 0
if kchmask > (kfqmask+kchmask)%11 then
partikkelset 1, kcount1, id
partikkelset 2, kcount1, id
kcount1 = (kcount1+1)%7
elseif kfqmask > (kfqmask*kchmask)%7 then
partikkelset 4, kcount2, id
kcount2 = (kcount2+kcount1)%6
endif

```

```
    endin  
  
    </CsInstruments>  
    <CsScore>  
    i1 0 30  
    </CsScore>  
    </CsoundSynthesizer>
```

## See Also

*partikkel*

## Credits

Author: Thom Johansen  
Author: Øyvind Brandtsegg  
May 2017

New in version 6.09

# partikkelset

partikkelset — Set mask index for a specific mask parameter of a running *partikkel* instance.

## Description

*partikkelset* is an opcode for setting the *partikkel* mask index for a specific parameter. Used together with *partikkelget*, it can be used to synchronize *partikkel* masking between several running instances of the *partikkel* opcode. It can also be used to set the internal mask index based on other processes, for example to create more complex masking patterns than is available with the regular grain masking system.

## Syntax

```
partikkelset kparameterindex, kmaskindex, iopcode_id
```

## Initialization

*iopcode\_id* -- the opcode id, linking an instance of *partikkel* to an instance of *partikkelsync*.

## Performance

*kparameterindex* -- mask parameter. Selection of the masking parameter for which to set the current mask index. The different parameters are identified as:

- 0: gain mask
- 1: pitch sweep start mask
- 2: pitch sweep end mask
- 3: fm modulation index mask
- 4: channel mask
- 5: waveform mix mask

*kmaskindex* -- value to set mask index to. Sets the current mask index for the parameter specified with *kparameterindex* in the *partikkel* instance identified with *iopcode\_id*.

## Example

Here is an example of the *partikkelget* and *partikkelset* opcodes. It uses the file *partikkelgetset.csd* [examples/partikkelgetset.csd].

**Example 665. Example of manipulating the internal mask index of *partikkel*, based on the value of other mask indices in the same *partikkel* instance.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out
-odac          ;;RT audio
; For Non-realtime ouput leave only the line below:
-o partikkel-getset.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

nchnls = 2
Odbfs = 1

giSine ftgen 0, 0, 65536, 10, 1
giCosine ftgen 0, 0, 8192, 9, 1, 1, 90
giSigmoidRise ftgen 0, 0, 8193, 19, 0.5, 1, 270, 1
giSigmoidFall ftgen 0, 0, 8193, 19, 0.5, 1, 90, 1

instr 1

; freqstart masking table, simple ascending semitones pattern
iwavfreqstarttab ftgentmp 0, 0, 32, -2, 0, 11,
    semitone(0), semitone(1), semitone(2), semitone(3),
    semitone(4), semitone(5), semitone(6), semitone(7),
    semitone(8), semitone(9), semitone(10), semitone(11)

iwavfreqendtab ftgentmp 0, 0, 32, -2, 0, 11,
    semitone(0), semitone(1), semitone(2), semitone(3),
    semitone(4), semitone(5), semitone(6), semitone(7),
    semitone(8), semitone(9), semitone(10), semitone(11)

; channel masking table, simple panning back and forth
ichannelmasks ftgentmp 0, 0, 32, -2, 0, 7, 0.0, 0.25, 0.5, 0.75, 1.0, 0.75, 0.5, 0.25

; init unused arate signals
awavfm = 0
asamplepos1 = 0
async = 0

; system
id = 1

a1,a2 partikkel 6, 0, -1, async, 0, -1, giSigmoidRise, giSigmoidFall,
    0.9, 0.5, 100, ampdbfs(-9), -1, 1, 0, iwavfreqstarttab, iwavfreqendtab, awavfm,
    -1, -1, giCosine, 1, 1, 1, ichannelmasks, 0,
    giSine, giSine, giSine, giSine,
    -1, asamplepos1, asamplepos1, asamplepos1, asamplepos1,
    440, 440, 440, 440, 100, id

outch 1, a1, 2, a2

; using partikkelget and partikkelset to make the channelmask and freqmask indices
; interact to create a more complex pattern
kfqmask partikkelget 1, id
kchmask partikkelget 4, id
kcount1 init 0
kcount2 init 0
if kchmask > (kfqmask+kchmask)%11 then
partikkelset 1, kcount1, id
partikkelset 2, kcount1, id
kcount1 = (kcount1+1)%7
elseif kfqmask > (kfqmask*kchmask)%7 then
partikkelset 4, kcount2, id
kcount2 = (kcount2+kcount1)%6
endif

```

```
endin  
  
</CsInstruments>  
<CsScore>  
i1 0 30  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*partikkel*

## Credits

Author: Thom Johansen  
Author: Øyvind Brandtsegg  
May 2017

New in version 6.09

# partikkelsync

*partikkelsync* — Outputs *partikkel*'s grain scheduler clock pulse and phase to synchronize several instances of the *partikkel* opcode to the same clock source.

## Description

*partikkelsync* is an opcode for outputting *partikkel*'s grain scheduler clock pulse and phase. *partikkelsync*'s output can be used to synchronize other instances of the *partikkel* opcode to the same clock.

## Syntax

```
async [,aphase] partikkelsync iopcode_id
```

## Initialization

*iopcode\_id* -- the opcode id, linking an instance of *partikkel* to an instance of *partikkelsync*.

## Performance

*async* -- trigger pulse signal. Outputs trigger pulses synchronized to a *partikkel* opcode's grain scheduler clock. One trigger pulse is generated for each grain started in the *partikkel* opcode with the same *opcode\_id*. The normal usage would be to send this signal to another *partikkel* opcode's *async* input to synchronize several instances of *partikkel*.

*aphase* -- clock phase. Outputs a linear ramping phase signal. Can be used e.g. for softsynchronization, or just as a phase generator ala *phasor*.

## Example

Here is an example of the *partikkelsync* opcodes. It uses the file *partikkelsync.csd* [examples/partikkel-sync.csd].

### Example 666. Example with soft sync of two *partikkel* generators.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out
-odac          ;;RT audio
; For Non-realtime output leave only the line below:
; -o partikkel_softsync.wav -W ;; for file output any platform
</CsOptions>

<CsInstruments>

sr = 44100
ksmps = 20
nchnls = 2

; Example by Oeyvind Brandtsegg 2007, revised 2008
```

```

giSine ftgen 0, 0, 65537, 10, 1
giCosine ftgen 0, 0, 8193, 9, 1, 1, 90
giSigmoRise ftgen 0, 0, 8193, 19, 0.5, 1, 270, 1 ; rising sigmoid
giSigmoFall ftgen 0, 0, 8193, 19, 0.5, 1, 90, 1 ; falling sigmoid

; *****
; example of soft synchronization of two partikkel instances
; *****
instr 1

/*score parameters*/
igrainrate = p4 ; grain rate
igrainsize = p5 ; grain size in ms
igrainFreq = p6 ; fundamental frequency of source waveform
iosc2Dev = p7 ; partikkel instance 2 grain rate deviation factor
iMaxSync = p8 ; max soft sync amount (increasing to this value during length of note)

/*overall envelope*/
iattack = 0.001
idecay = 0.2
isustain = 0.7
irelease = 0.2
amp linsegr 0, iattack, 1, idecay, isustain, 1, isustain, irelease, 0

kgrainfreq = igrainrate ; grains per second
kdistribution = 0 ; periodic grain distribution
idisttab = -1 ; (default) flat distribution used
; for grain distribution

async = 0 ; no sync input
kenv2amt = 0 ; no secondary enveloping
ienv2tab = -1 ; default secondary envelope (flat)
ienv_attack = giSigmoRise ; default attack envelope (flat)
ienv_decay = giSigmoFall ; default decay envelope (flat)
ksustain_amount = 0.3 ; time (in fraction of grain dur) at
; sustain level for each grain
ka_d_ratio = 0.2 ; balance between attack and decay time
kduration = igrainsize ; set grain duration in ms
kamp = 0.2*Odbfs ; amp
igainmasks = -1 ; (default) no gain masking
kwavfreq = igrainFreq ; fundamental frequency of source waveform
ksweepshape = 0 ; shape of frequency sweep (0=no sweep)
iwavfreqstarttab = -1 ; default frequency sweep start
; (value in table = 1, which give
; no frequency modification)
iwavfreqendtab = -1 ; default frequency sweep end
; (value in table = 1, which give
; no frequency modification)

awavfm = 0 ; no FM input
ifmamptab = -1 ; default FM scaling (=1)
kfmenv = -1 ; default FM envelope (flat)
icosine = giCosine ; cosine ftable
kTrainCps = kgrainfreq ; set trainlet cps equal to grain
; rate for single-cycle trainlet in
; each grain
knumpartials = 3 ; number of partials in trainlet
kchroma = 1 ; balance of partials in trainlet
ichannelmasks = -1 ; (default) no channel masking,
; all grains to output 1
krandommask = 0 ; no random grain masking
kwaveform1 = giSine ; source waveforms
kwaveform2 = giSine ;
kwaveform3 = giSine ;
kwaveform4 = giSine ;
iwaveamptab = -1 ; mix of 4 source waveforms and
; trainlets (set to default)
asamplepos1 = 0 ; phase offset for reading source waveform

```

```

asamplepos2 = 0 ;
asamplepos3 = 0 ;
asamplepos4 = 0 ;
kwavekey1 = 1 ; original key for source waveform
kwavekey2 = 1 ;
kwavekey3 = 1 ;
kwavekey4 = 1 ;
imax_grains = 100 ; max grains per k period
iopcode_id = 1 ; id of opcode, linking partikkel
                    ; to partikkelsync

a1 partikkel kgrainfreq, kdistribution, idisttab, async, kenv2amt, \
    ienv2tab, ienv_attack, ienv_decay, ksustain_amount, ka_d_ratio, \
    kduration, kamp, igainmasks, kwavfreq, ksweepshape, \
    iwavfreqstarttab, iwavfregendtab, awavfm, ifmamptab, kfmenv, \
    icosine, kTrainCps, knumpartials, kchroma, ichannelmasks, \
    krandommask, kwaveform1, kwaveform2, kwaveform3, kwaveform4, \
    iwaveamptab, asamplepos1, asamplepos2, asamplepos3, asamplepos4, \
    kwavekey1, kwavekey2, kwavekey3, kwavekey4, imax_grains, iopcode_id

async1 partikkelsync iopcode_id ; clock pulse output of the
                                ; partikkel instance above
ksyncGravity line 0, p3, iMaxSync ; strength of synchronization
aphase2 init 0
asyncPolarity limit (int(aphase2*2)*2)-1, -1, 1
; use the phase of partikkelsync instance 2 to find sync
; polarity for partikkel instance 2.
; If the phase of instance 2 is less than 0.5, we want to
; nudge it down when synchronizing,
; and if the phase is > 0.5 we want to nudge it upwards.
async1 = async1*ksyncGravity*asyncPolarity ; prepare sync signal
                                ; with polarity and strength

kgrainfreq2 = igrainrate * iosc2Dev ; grains per second for second partikkel instance
iopcode_id2 = 2
a2 partikkel kgrainfreq2, kdistribution, idisttab, async1, kenv2amt, \
    ienv2tab, ienv_attack, ienv_decay, ksustain_amount, ka_d_ratio, \
    kduration, kamp, igainmasks, kwavfreq, ksweepshape, \
    iwavfreqstarttab, iwavfregendtab, awavfm, ifmamptab, kfmenv, \
    icosine, kTrainCps, knumpartials, kchroma, ichannelmasks, \
    krandommask, kwaveform1, kwaveform2, kwaveform3, kwaveform4, \
    iwaveamptab, asamplepos1, asamplepos2, asamplepos3, \
    asamplepos4, kwavekey1, kwavekey2, kwavekey3, kwavekey4, \
    imax_grains, iopcode_id2

async2, aphase2 partikkelsync iopcode_id2
; clock pulse and phase
; output of the partikkel instance above,
; we will only use the phase

outs a1*amp, a2*amp

endin

</CsInstruments>
<CsScore>

/*score parameters
igrainrate = p4 ; grain rate
igrainsize = p5 ; grain size in ms
igrainFreq = p6 ; frequency of source wave within grain
iosc2Dev = p7 ; partikkel instance 2 grain rate deviation factor
iMaxSync = p8 ; max soft sync amount (increasing to this value during length of note)
*/
; GrRate GrSize GrFund Osc2Dev MaxSync

i1 0 10 2 20 880 1.3 0.3

```

```
s
i1 0 10 5 20 440 0.8 0.3
s
i1 0 6 55 15 660 1.8 0.45
s
i1 0 6 110 10 440 0.6 0.6
s
i1 0 6 220 3 660 2.6 0.45
s
i1 0 6 220 3 660 2.1 0.45
s
i1 0 6 440 3 660 0.8 0.22
s

e

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*partikkel*

## Credits

Author: Thom Johansen  
Author: Torgeir Strand Henriksen  
Author: Øyvind Brandtsegg  
April 2007

New in version 5.06

# passign

passign — Assigns a range of p-fields to ivariables.

## Description

Assigns a range of p-fields to ivariables.

## Syntax

```
ivar1, ... passign [istart]
```

## Initialisation

The optional argument *istart* gives the index of the first p-field to assign. The default value is 1, corresponding to the instrument number.

One of the variables can be a string variable, in which case it is assigned the only string parameter, if there is one, or an error otherwise.

## Performance

*passign* transfers the instrument p-fields to instrument variables, starting with the one identified by the *istart* argument. There should not be more variables than p-fields, but there may be fewer.

## See Also

*assign*,

## Example

Here is an example of the *passign* opcode. It uses the file *passign.csd* [examples/passign.csd].

**Example 667. A variant of toot8.csd that uses passign.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if real audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o passign.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 8
```

```
idur, iamp, iskiptime, iattack, irelease, irvbtime, irvbgain passign 3

kamp      linen      iamp, iattack, idur, irelease
asig      soundin    "fox.wav", iskiptime
arampsig  =          kamp * asig
aeffect   reverb     asig, irvbtime
arvbretrn =         aeffect * irvbgain
;mix dry & wet signals
          outs      arampsig + arvbretrn, arampsig + arvbretrn

          endin

</CsInstruments>
<CsScore>

;ins strt dur  amp  skip atk  rel      rvbt rvgain
i8  0   4   .3  0   .03  .1      1.5  .3
i8  4   4   .3  1.6 .1   .1      3.1  .7
i8  8   4   .3  0   .5   .1      2.1  .2
i8 12   4   .4  0   .01  .1      1.1  .1
i8 16   4   .5  0.1 .01  .1      0.1  .1

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitth  
University of Bath, Codemist Ltd.  
Bath, UK  
December 2009

New in version 5.12

# paulstretch

paulstretch — Extreme time-stretching algorithm by Nasca Octavian Paul.

## Description

The *paulstretch* opcode is a lightweight implementation of the PaulStretch time-stretching algorithm by Nasca Octavian Paul. It is ideal for timestretching a signal by very large amounts.

The Paulstretch algorithm works very similarly to other STFT-based timestretching methods, using overlap-add with a hanning window. Unique to paulstretch are the use of scrambled phase and very large window sizes (specified in seconds).

## Syntax

```
asig paulstretch istretch, iwindowsize, ift
```

## Initialization

*istretch* -- Stretch factor.

*iwindowsize* -- Window size, in seconds.

*ift* -- source signal function table. Deferred-allocation tables (see *GEN01*) are accepted, but the opcode expects a mono source.

## Examples

Here is an example of the paulstretch opcode. It uses the file *paulstretch.csd* [examples/paulstretch.csd]

### Example 668. Example of the paulstretch opcode.

```
<CsoundSynthesizer>
<CsOptions>
-o paulstretch.wav -W
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 100
0dbfs = 1
nchnls = 2

giwav ftgen 0, 0, 0, 1, "fox.wav", 0, 0, 1

instr 1
aout paulstretch 10, 0.2, giwav
outs aout, aout
endin

</CsInstruments>
<CsScore>
i1 0 30
```

```
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Author: Paul Batchelor  
May 2016

# pcauchy

pcauchy — Cauchy distribution random number generator (positive values only).

## Description

Cauchy distribution random number generator (positive values only). This is an x-class noise generator.

## Syntax

```
ares pcauchy kalpha
ires pcauchy kalpha
kres pcauchy kalpha
```

## Performance

*pcauchy kalpha* -- controls the spread from zero (big kalpha = big spread). Outputs positive numbers only.

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the pcauchy opcode. It uses the file *pcauchy.csd* [examples/pcauchy.csd].

### Example 669. Example of the pcauchy opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if real audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pcauchy.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; every run time same values

kalpha pcauchy 1000
printk .2, kalpha ; look
```

```
aout oscili 0.8, 440+kalpha, 1 ; & listen
outs aout, aout
endin

instr 2 ; every run time different values

seed 0
kalpha pcauchy 1000
printk .2, kalpha ; look
aout oscili 0.8, 440+kalpha, 1 ; & listen
outs aout, aout
endin
</CsInstruments>
<CsScore>
; sine wave
f 1 0 16384 10 1

i 1 0 2
i 2 3 2
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include a line like this:

```
i 1 time 0.00033: 10.48851
i 1 time 0.20033: 0.29508
i 1 time 0.40033: 1.75214
i 1 time 0.60033: 22.88281
i 1 time 0.80033: 16.06435
i 1 time 1.00000: 0.43110
i 1 time 1.20033: 16.51694
i 1 time 1.40033: 2.98797
i 1 time 1.60033: 1.32767
i 1 time 1.80000: 17.94039
i 1 time 2.00000: 4.85994
Seeding from current time 1526147515
i 2 time 3.00033: 0.89797
i 2 time 3.20033: 9.19447
i 2 time 3.40033: 0.73053
i 2 time 3.60000: 7.43371
i 2 time 3.80033: 5.87640
i 2 time 4.00000: 0.80456
i 2 time 4.20000: 4.50637
i 2 time 4.40033: 2.08145
i 2 time 4.60033: 13.47125
i 2 time 4.80033: 3.16399
i 2 time 5.00000: 11.05428
```

## See Also

*seed, betarand, bexprnd, cauchy, exprand, gauss, linrand, poisson, trirand, unirand, weibull*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

# pchbend

pchbend — Get the current pitch-bend value for this channel.

## Description

Get the current pitch-bend value for this channel.

## Syntax

```
ibend pchbend [imin] [, imax]
kbend pchbend [imin] [, imax]
```

## Initialization

*imin*, *imax* (optional) -- set minimum and maximum limits on values obtained

## Performance

Get the current pitch-bend value for this channel. Note that this access to pitch-bend data is independent of the MIDI pitch, enabling the value here to be used for any arbitrary purpose.

## Examples

Here is an example of the pchbend opcode. It uses the file *pchbend.csd* [examples/pchbend.csd].

### Example 670. Example of the pchbend opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -Q1 -Ma ;;realtime audio out and midi in (on all inputs) and out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pchbend.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;use external midi keyboard

icps cpsmidi
kbnd pchbend 0, 100 ;one octave lower and higher
kenv linsegr 0,.001, 1, .1, 0 ;amplitude envelope
asig pluck .8 * kenv, icps+kbnd, 440, 0, 1
outs asig, asig
```

```
    endin
  </CsInstruments>
<CsScore>

  f 0 30 ;runs 30 seconds

</CsScore>
</CsoundSynthesizer>
```

## See Also

*aftouch, ampmidi, cpsmidi, cpsmidib, midictrl, notnum, octmidi, octmidib, pchmidi, pchmidib, veloc*

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

# pchmidi

pchmidi — Get the note number of the current MIDI event, expressed in pitch-class units.

## Description

Get the note number of the current MIDI event, expressed in pitch-class units.

## Syntax

```
ipch pchmidi
```

## Performance

Get the note number of the current MIDI event, expressed in pitch-class units for local processing.



### pchmidi vs. pchmidinn

The *pchmidi* opcode only produces meaningful results in a Midi-activated note (either real-time or from a Midi score with the -F flag). With *pchmidi*, the Midi note number value is taken from the Midi event that is internally associated with the instrument instance. On the other hand, the *pchmidinn* opcode may be used in any Csound instrument instance whether it is activated from a Midi event, score event, line event, or from another instrument. The input value for *pchmidinn* might for example come from a p-field in a textual score or it may have been retrieved from the real-time Midi event that activated the current note using the *notnum* opcode.

## Examples

Here is an example of the pchmidi opcode. It uses the file *pchmidi.csd* [examples/pchmidi.csd].

### Example 671. Example of the pchmidi opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages  MIDI in
-odac      -iadc      -d      -M0      ;;RT audio I/O with MIDI in
; For Non-realtime ouput leave only the line below:
; -o pchmidi.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
```

```
instr 1
; This example expects MIDI note inputs on channel 1
i1 pchmidi

print i1
endin

</CsInstruments>
<CsScore>

;Dummy f-table to give time for real-time MIDI events
f 0 8000
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*aftouch, ampmidi, cpsmidi, cpsmidib, midictrl, notnum, octmidi, octmidib, pchbend, pchmidib, veloc, cpsmidim, octmidim, pchmidim*

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

Example written by Kevin Conder.

# pchmidib

pchmidib — Get the note number of the current MIDI event and modify it by the current pitch-bend value, express it in pitch-class units.

## Description

Get the note number of the current MIDI event and modify it by the current pitch-bend value, express it in pitch-class units.

## Syntax

```
ipch pchmidib [irange]
kpch pchmidib [irange]
```

## Initialization

*irange* (optional) -- the pitch bend range in semitones

## Performance

Get the note number of the current MIDI event, modify it by the current pitch-bend value, and express the result in pitch-class units. Available as an i-time value or as a continuous k-rate value.

## Examples

Here is an example of the pchmidib pchmidib. It uses the file *pchmidib.csd* [examples/pchmidib.csd].

### Example 672. Example of the pchmidib pchmidib.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages  MIDI in
-odac          -iadc      -d          -M0    ;;RT audio I/O with MIDI in
; For Non-realtime ouput leave only the line below:
; -o pchmidib.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; This example expects MIDI note inputs on channel 1
i1 pchmidib
```

```
    print i1
  endin

</CsInstruments>
<CsScore>

;Dummy f-table to give time for real-time MIDI events
f 0 8000
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*aftouch, ampmidi, cpsmidi, cpsmidib, midictrl, notnum, octmidi, octmidib, pchbend, pchmidi, veloc*

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

Example written by Kevin Conder.

# pchmidinn

`pchmidinn` — Converts a Midi note number value to octave point pitch-class units.

## Description

Converts a Midi note number value to octave point pitch-class units.

## Syntax

```
pchmidinn (MidiNoteNumber) (init- or control-rate args only)
```

where the argument within the parentheses may be a further expression.

## Performance

`pchmidinn` is a function that takes an i-rate or k-rate value representing a Midi note number and returns the equivalent pitch value in Csound's octave point pitch-class format. This conversion assumes that Middle C (8.00 in *pch*) is Midi note number 60. Midi note number values are typically integers in the range from 0 to 127 but fractional values or values outside of this range will be interpreted consistently.



### **pchmidinn vs. pchmidi**

The `pchmidinn` opcode may be used in any Csound instrument instance whether it is activated from a Midi event, score event, line event, or from another instrument. The input value for `pchmidinn` might for example come from a p-field in a textual score or it may have been retrieved from the real-time Midi event that activated the current note using the `notnum` opcode. You must specify an i-rate or k-rate expression for the Midi note number that is to be converted. On the other hand, the `pchmidi` opcode only produces meaningful results in a Midi-activated note (either real-time or from a Midi score with the -F flag). With `pchmidi`, the Midi note number value is taken from the Midi event associated with the instrument instance, and no location or expression for this value may be specified.

`pchmidinn` and its related opcodes are really *value converters* with a special function of manipulating pitch data.

Data concerning pitch and frequency can exist in any of the following forms:

**Table 19. Pitch and Frequency Values**

Name	Abbreviation
octave point pitch-class (8ve.pc)	pch
octave point decimal	oct
cycles per second	cps
Midi note number (0-127)	midinn

The first two forms consist of a whole number, representing octave registration, followed by a specially interpreted fractional part. For *pch*, the fraction is read as two decimal digits representing the 12 equal-tempered pitch classes from .00 for C to .11 for B. For *oct*, the fraction is interpreted as a true decimal fractional part of an octave. The two fractional forms are thus related by the factor 100/12. In both forms,

the fraction is preceded by a whole number octave index such that 8.00 represents Middle C, 9.00 the C above, etc. Midi note number values range between 0 and 127 (inclusively) with 60 representing Middle C, and are usually whole numbers. Thus A440 can be represented alternatively by 440 (*cps*), 69 (*midinn*), 8.09 (*pch*), or 8.75 (*oct*). Microtonal divisions of the *pch* semitone can be encoded by using more than two decimal places.

The mnemonics of the pitch conversion units are derived from morphemes of the forms involved, the second morpheme describing the source and the first morpheme the object (result). Thus *cpspch*(8.09) will convert the pitch argument 8.09 to its *cps* (or Hertz) equivalent, giving the value of 440. Since the argument is constant over the duration of the note, this conversion will take place at i-time, before any samples for the current note are produced.

By contrast, the conversion *cpsoct*(8.75 + k1) which gives the value of A440 transposed by the octave interval *k1*. The calculation will be repeated every *k*-period since that is the rate at which *k1* varies.



## Note

The conversion from *pch*, *oct*, or *midinn* into *cps* is not a linear operation but involves an exponential process that could be time-consuming when executed repeatedly. Csound now uses a built-in table lookup to do this efficiently, even at audio rates. Because the table index is truncated without interpolation, pitch resolution when using one of these opcodes is limited to 8192 discrete and equal divisions of the octave, and some pitches of the standard 12-tone equally-tempered scale are very slightly mistuned (by at most 0.15 cents).

## Examples

Here is an example of the *pchmidinn* opcode. It uses the file *cpsmidinn.csd* [examples/cpsmidinn.csd].

### Example 673. Example of the *pchmidinn* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
; Prints a table showing the equivalents of all Midi
; note numbers from 0-127 in cycles-per-second,
; octave.decimal, and octave.pitchclass units.

<CsOptions>
; Select audio/midi flags here according to platform.
; This example produces no audio, so we render in
; non-realttime and turn off sound to disk:
-n
</CsOptions>
<CsInstruments>

instr 1
; i-time loop to print conversion table
imidiNN = 0
loop1:
  icps = cpsmidinn(imidiNN)
  ioct = octmidinn(imidiNN)
  ipch = pchmidinn(imidiNN)

  print imidiNN, icps, ioct, ipch

  imidiNN = imidiNN + 1
if (imidiNN < 128) igoto loop1
```

```
    endin

    instr 2
      ; test k-rate converters
      kMiddleC = 60
      kcps = cpsmidinn(kMiddleC)
      koct = octmidinn(kMiddleC)
      kpch = pchmidinn(kMiddleC)

      printks "%d %f %f %f\n", 1.0, kMiddleC, kcps, koct, kpch
    endin

    </CsInstruments>
    <CsScore>
    i1 0 0
    i2 0 0.1
    e

    </CsScore>
  </CsoundSynthesizer>
```

## See Also

*cpsmidinn, octmidinn, pchmidi, notnum, cpspch, cpsoct, octcps, octpch, pchoct*

## Credits

Derived from original value converters by Barry Vercoe.

New in version 5.07

# pchoct

pchoct — Converts an octave-point-decimal value to pitch-class.

## Description

Converts an octave-point-decimal value to pitch-class.

## Syntax

`pchoct (oct) (init- or control-rate args only)`

where the argument within the parentheses may be a further expression.

## Performance

*pchoct* and its related opcodes are really *value converters* with a special function of manipulating pitch data.

Data concerning pitch and frequency can exist in any of the following forms:

**Table 20. Pitch and Frequency Values**

Name	Abbreviation
octave point pitch-class (8ve.pc)	pch
octave point decimal	oct
cycles per second	cps
Midi note number (0-127)	midinn

The first two forms consist of a whole number, representing octave registration, followed by a specially interpreted fractional part. For *pch*, the fraction is read as two decimal digits representing the 12 equal-tempered pitch classes from .00 for C to .11 for B. For *oct*, the fraction is interpreted as a true decimal fractional part of an octave. The two fractional forms are thus related by the factor 100/12. In both forms, the fraction is preceded by a whole number octave index such that 8.00 represents Middle C, 9.00 the C above, etc. Midi note number values range between 0 and 127 (inclusively) with 60 representing Middle C, and are usually whole numbers. Thus A440 can be represented alternatively by 440 (*cps*), 69 (*midinn*), 8.09 (*pch*), or 8.75 (*oct*). Microtonal divisions of the *pch* semitone can be encoded by using more than two decimal places.

The mnemonics of the pitch conversion units are derived from morphemes of the forms involved, the second morpheme describing the source and the first morpheme the object (result). Thus *cpspch*(8.09) will convert the pitch argument 8.09 to its *cps* (or Hertz) equivalent, giving the value of 440. Since the argument is constant over the duration of the note, this conversion will take place at i-time, before any samples for the current note are produced.

By contrast, the conversion *cpsoct*(8.75 + k1) which gives the value of A440 transposed by the octave interval *k1*. The calculation will be repeated every k-period since that is the rate at which *k1* varies.



### Note

The conversion from *pch*, *oct*, or *midinn* into *cps* is not a linear operation but involves an exponential process that could be time-consuming when executed repeatedly. Csound now

uses a built-in table lookup to do this efficiently, even at audio rates. Because the table index is truncated without interpolation, pitch resolution when using one of these opcodes is limited to 8192 discrete and equal divisions of the octave, and some pitches of the standard 12-tone equally-tempered scale are very slightly mistuned (by at most 0.15 cents).

## Examples

Here is an example of the `pchoct` opcode. It uses the file `pchoct.csd` [examples/pchoct.csd].

### Example 674. Example of the `pchoct` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o pchoct.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Convert an octave-point-decimal value into a
; pitch-class value.
ioct = 8.75
ipch = pchoct(ioct)

print ipch
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include a line like this:

```
instr 1: ipch = 8.090
```

## See Also

*cpsoct*, *cpspch*, *octcps*, *octpch*, *cpsmidinn*, *octmidinn*, *pchmidinn*

## Credits

Example written by Kevin Conder.

# pchtom

pchtom — Convert pch to midi note number

## Description

Convert pch to midi note number. **pch** representation has the form Octave.pitchclass, pitchclass being a number between 00 and 12.

pch	midi	note name
8.09	69	4A
8.00	60	4C

## Syntax

```
imidi pchtom ipch  
kmidi pchtom kpch
```

## Performance

*kpch / ipch* -- pitch represented as Octave.pitchclass

*kmidi / imidi* -- midi note number



### Note

Use *pchmidinn* to perform the opposite operation of this opcode.

## Examples

Here is an example of the pchtom opcode. It uses the file *pchtom.csd* [examples/pchtom.csd].

### Example 675. Example of the pchtom opcode.

```
<CsoundSynthesizer>  
<CsOptions>  
</CsOptions>  
<CsInstruments>  
  
sr = 44100  
ksmps = 128  
nchnls = 2  
0dbfs = 1.0  
  
; Show some conversions, both at i- and at k-time  
  
instr 1  
  imidi = pchtom(8.09)  
  print imidi
```

```
kidx init 0

kpch = 8 + kidx / 100
kmidi = pchtom(kpch)
kidx += 1
printf "kpch: %f   kmidi: %f\n", kidx+1, kpch, kmidi

if kidx >= 12 then
  turnoff
endif

endin

</CsInstruments>
<CsScore>
i 1 0 1

</CsScore>
</CsoundSynthesizer>
```

## See Also

*pchoct, pchmidinn, cpsmidinn, ftom, mton*

## Credits

By: Eduardo Moguillansky 2017

# pconvolve

convolve — Convolution based on a uniformly partitioned overlap-save algorithm

## Description

Convolution based on a uniformly partitioned overlap-save algorithm. Compared to the *convolve* opcode, *pconvolve* has these benefits:

- small delay
- possible to run in real-time for shorter impulse files
- no pre-process analysis pass
- can often render faster than *convolve*

## Syntax

```
ar1 [, ar2] [, ar3] [, ar4] pconvolve ain, ifilcod [, ipartitionsizes, ichannel]
```

## Initialization

*ifilcod* -- integer or character-string denoting an impulse response soundfile. Multichannel files are supported, the file must have the same sample-rate as the orc. [Note: *cvanal* files cannot be used!] Keep in mind that longer files require more calculation time [and probably larger partition sizes and more latency]. At current processor speeds, files longer than a few seconds may not render in real-time.

*ipartitionsizes* (optional, defaults to the output buffersize [-b]) -- the size in samples of each partition of the impulse file. This is the parameter that needs tweaking for best performance depending on the impulse file size. Generally, a small size means smaller latency but more computation time. If you specify a value that is not a power-of-2 the opcode will find the next power-of-2 greater and use that as the actual partition size.

*ichannel* (optional) -- which channel to use from the impulse response data file.

## Performance

*ain* -- input audio signal.

The overall latency of the opcode can be calculated as such [assuming *ipartitionsizes* is a power of 2]

$$latency = (ksmps < ipartitionsizes ? ipartitionsizes + ksmps : ipartitionsizes) / sr$$

## Examples

Instrument 1 shows an example of real-time convolution.

Instrument 2 shows how to do file-based convolution with a 'look ahead' method to remove all delay.



### NOTE

You can download impulse response files from [noisevault.com](http://noisevault.com) or replace the filenames with your own impulse files.

Here is an example of the pconvolve opcode. It uses the file *pconvolve.csd* [examples/pconvolve.csd].

### Example 676. Example of the pconvolve opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
-iadc   ;;uncomment -iadc if real audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pconvolve.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kmix = .5 ; Wet/dry mix. Vary as desired.
kvol = .05*kmix ; Overall volume level of reverb. May need to adjust
; when wet/dry mix is changed, to avoid clipping.

; do some safety checking to make sure we the parameters a good
kmix = (kmix < 0 || kmix > 1 ? .5 : kmix)
kvol = (kvol < 0 ? 0 : .5*kvol*kmix)

; size of each convolution partion -- for best performance, this parameter needs to be tweaked
ipartitionsize = p4

; calculate latency of pconvolve opcode
idel = (ksmps < ipartitionsize ? ipartitionsize + ksmps : ipartitionsize)/sr
prints "Convoluting with a latency of %f seconds%n", idel

; actual processing
al, ar ins ;get live input
awetl, awetr pconvolve kvol*(al+ar), "kickroll.wav", ipartitionsize
; Delay dry signal, to align it with the convoled sig
adryl delay (1-kmix)*al, idel
adryr delay (1-kmix)*al, idel
outs adryl+awetl, adryr+awetr

endin

instr 2

imix = 0.5 ; Wet/dry mix. Vary as desired.
ivol = .05*imix ; Overall volume level of reverb. May need to adjust
; when wet/dry mix is changed, to avoid clipping.
ipartitionsize = 1024 ; size of each convolution partion
idel = (ksmps < ipartitionsize ? ipartitionsize + ksmps : ipartitionsize)/sr ; latency of pconvolve c

kcount init idel*kr
; since we are using a soundin [instead of ins] we can
; do a kind of "look ahead" by looping during one k-pass
; without output, creating zero-latency
loop:
asig soundin p4, 0
awetl, awetr pconvolve ivol*(asig),"rv_stereo.wav", ipartitionsize
adry delay (1-imix)*asig,idel ; Delay dry signal, to align it with

```

```
    kcount = kcount - 1
  if kcount > 0 kgoto loop
    outs awetl+adry, awetr+adry

  endin
</CsInstruments>
<CsScore>

  i 1 0 20 1024 ;play live for 20 seconds

  i 2 20 5 "fox.wav"
  i 2 25 5 "flute.aiff"

  e
</CsScore>
</CsoundSynthesizer>
```

## See also

*convolve*, *dconv*.

## Credits

Author: Matt Ingalls  
2004

# pcount

pcount — Returns the number of pfields belonging to a note event.

## Description

*pcount* returns the number of pfields belonging to a note event.

## Syntax

icount **pcount**

## Initialization

*icount* - stores the number of pfields for the current note event.



### Note

Note that the reported number of pfields is not necessarily what's explicitly written in the score, but the pfields available to the instrument through mechanisms like *pfield carry*.

## Examples

Here is an example of the pcount opcode. It uses the file *pcount.csd* [examples/pcount.csd].

### Example 677. Example of the pcount opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac ;;realtime audio
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pcount.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

; This UDO returns a pfield value but returns 0 if it does not exist.
opcode mypvalue, i, i
  index xin
  inum pcount
  if (index > inum) then
    iout = 0.0
  else
    iout pindex index
  endif
```

```

xout iout
endop

; Envelope UDO that reads parameters from a flexible number of pfields
; Syntax:  kenv flexlinseg ipstart
;         ipstart is the first pfield of the envelope
;         parameters. Reads remaining pfields (up to 21 of them).
;         kenv is the output envelope.

opcode flexlinseg, k, i
  ipstart xin

  iep1 mypvalue ipstart
  iep2 mypvalue ipstart + 1
  iep3 mypvalue ipstart + 2
  iep4 mypvalue ipstart + 3
  iep5 mypvalue ipstart + 4
  iep6 mypvalue ipstart + 5
  iep7 mypvalue ipstart + 6
  iep8 mypvalue ipstart + 7
  iep9 mypvalue ipstart + 8
  iepa mypvalue ipstart + 9
  iepb mypvalue ipstart + 10
  iepc mypvalue ipstart + 11
  iepd mypvalue ipstart + 12
  iepe mypvalue ipstart + 13
  iepf mypvalue ipstart + 14
  iepg mypvalue ipstart + 15
  ieph mypvalue ipstart + 16
  iepi mypvalue ipstart + 17
  iepj mypvalue ipstart + 18
  iepk mypvalue ipstart + 19
  iepl mypvalue ipstart + 20

  kenv linseg iep1, iep2, iep3, iep4, iep5, iep6, iep7, iep8, \
        iep9, iepa, iepb, iepc, iepd, iepe, iepf, iepg, \
        ieph, iepi, iepj, iepk, iepl

xout kenv
endop

instr 1
; This instrument only requires 3 pfields but can accept up to 24.
; (You will still get warnings about more than 3).
kenv flexlinseg 4 ; envelope params start at p4
aout oscili kenv*.5, 256, 1
outs aout, aout

endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1

i1 0 2 0.0 1.0 1.0 1.0 0.0
i1 2 2 0.0 0.1 1.0 1.0 1.0 0.1 0.0
i1 4 2 0.0 0.5 0.0 ; one problem is that "missing" pfields carry
i1 6 2 0.0 0.5 0.0 ! ; now we can fix this problem with !
i1 8 10 0.0 3.0 1.0 0.3 0.1 0.3 1.0 0.3 0.1 0.3 1.0 0.3 0.1 0.8 0.9 5.0 0.0

e
</CsScore>
</CsSoundSynthesizer>

```

The example will produce the following output:

```

WARNING: instr 1 uses 3 p-fields but is given 8
B 0.000 .. 2.000 T 2.000 TT 2.000 M: 0.49966 0.49966

```

```
WARNING: instr 1 uses 3 p-fields but is given 10
B 2.000 .. 4.000 T 4.000 TT 4.000 M: 0.50000 0.50000
WARNING: instr 1 uses 3 p-fields but is given 10
B 4.000 .. 6.000 T 6.000 TT 6.000 M: 0.49943 0.49943
WARNING: instr 1 uses 3 p-fields but is given 6
B 6.000 .. 8.000 T 8.000 TT 8.000 M: 0.00000 0.00000
WARNING: instr 1 uses 3 p-fields but is given 20
B 8.000 .. 18.000 T 18.000 TT 18.000 M: 0.49994 0.49994
```

The warnings occur because pfields are not used explicitly by the instrument.

## See Also

*pindex*

## Credits

Example by: Anthony Kozar

Dec. 2006

# pdclip

pdclip — Performs linear clipping on an audio signal or a phasor.

## Description

The *pdclip* opcode allows a percentage of the input range of a signal to be clipped to fullscale. It is similar to simply multiplying the signal and limiting the range of the result, but *pdclip* allows you to think about how much of the signal range is being distorted instead of the scalar factor and has a offset parameter for assymetric clipping of the signal range. *pdclip* is also useful for remapping phasors for phase distortion synthesis.

## Syntax

```
aout pdclip ain, kWidth, kCenter [, ibipolar [, ifullscale]]
```

## Initialization

*ibipolar* -- an optional parameter specifying either unipolar (0) or bipolar (1) mode. Defaults to unipolar mode.

*ifullscale* -- an optional parameter specifying the range of input and output values. The maximum will be *ifullscale*. The minimum depends on the mode of operation: zero for unipolar or *-ifullscale* for bipolar. Defaults to 1.0 -- you should set this parameter to the maximum expected input value.

## Performance

*ain* -- the input signal to be clipped.

*aout* -- the output signal.

*kWidth* -- the percentage of the signal range that is clipped (must be between 0 and 1).

*kCenter* -- an offset for shifting the unclipped window of the signal higher or lower in the range (essentially a DC offset). Values should be in the range [-1, 1] with a value of zero representing no shift (regardless of whether bipolar or unipolar mode is used).

The *pdclip* opcode performs linear clipping on the input signal *ain*. *kWidth* specifies the percentage of the signal range that is clipped. The rest of the input range is mapped linearly from zero to *ifullscale* in unipolar mode and from *-ifullscale* to *ifullscale* in bipolar mode. When *kCenter* is zero, equal amounts of the top and bottom of the signal range are clipped. A negative value shifts the unclipped range more towards the bottom of the input range and a positive value shifts it more towards the top. *ibipolar* should be 1 for bipolar operation and 0 for unipolar mode. The default is unipolar mode (*ibipolar* = 0). *ifullscale* sets the maximum amplitude of the input and output signals (defaults to 1.0).

This amounts to waveshaping the input with the following transfer function (normalized to *ifullscale*=1.0 in bipolar mode):

1| \_\_\_\_\_ x-axis is input range, y-axis is output  
 | /  
 | / width of clipped region is 2\*kWidth

```

-1  | 1  width of unclipped region is 2*(1 - kWidth)
----- kCenter shifts the unclipped region
      |
      | left or right (up to kWidth)
      |
      |
----- |-1

```

Bipolar mode can be used for direct, linear distortion of an audio signal. Alternatively, unipolar mode is useful for modifying the output of a phasor before it is used to index a function table, effectively making this a phase distortion technique.

## See Also

*pdhalf*, *pdhalfy*, *limit*, *clip*, *distort1*

## Examples

Here is an example of the `pdclip` opcode. It uses the file `pdclip.csd` [examples/pdclip.csd].

### Example 678. Example of the `pdclip` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o abs.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; test instrument for pdclip opcode
instr 3

  idur  = p3
  iamp  = p4
  ifreq = p5
  ifn   = p6

  kenv  linseg  0, .05, 1.0, idur - .1, 1.0, .05, 0
  aosc  oscil   1.0, ifreq, ifn

  kmod  expseg  0.00001, idur, 1.0
  aout  pdclip  aosc, kmod, 0.0, 1.0

  out   kenv*aout*iamp
endin

</CsInstruments>
<CsScore>
f1 0 16385 10 1
f2 0 16385 10 1 .5 .3333 .25 .5

; pdclipped sine wave
i3 0 3 15000 440 1
i3 + 3 15000 330 1

```

```
i3 + 3 15000 220 1
s

; pdclipped composite wave
i3 0 3 15000 440 2
i3 + 3 15000 330 2
i3 + 3 15000 220 2
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Anthony Kozar  
January 2008

New in Csound version 5.08

# pdhalf

pdhalf — Distorts a phasor for reading the two halves of a table at different rates.

## Description

The *pdhalf* opcode is designed to emulate the "classic" phase distortion synthesis method of the Casio CZ-series of synthesizers from the mid-1980's. This technique reads the first and second halves of a function table at different rates in order to warp the waveform. For example, *pdhalf* can smoothly transform a sine wave into something approximating the shape of a saw wave.

## Syntax

```
aout pdhalf ain, kShapeAmount [, ibipolar [, ifullscale]]
```

## Initialization

*ibipolar* -- an optional parameter specifying either unipolar (0) or bipolar (1) mode. Defaults to unipolar mode.

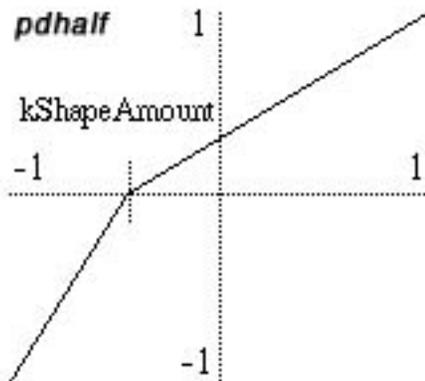
*ifullscale* -- an optional parameter specifying the range of input and output values. The maximum will be *ifullscale*. The minimum depends on the mode of operation: zero for unipolar or *-ifullscale* for bipolar. Defaults to 1.0 -- you should set this parameter to the maximum expected input value.

## Performance

*ain* -- the input signal to be distorted.

*aout* -- the output signal.

*kShapeAmount* -- the amount of distortion applied to the input. Must be between negative one and one (-1 to 1). An amount of zero means no distortion.



Transfer function created by *pdhalf* and a negative *kShapeAmount*.

The *pdhalf* opcode calculates a transfer function that is composed of two linear segments (see the graph). These segments meet at a "pivot point" which always lies on the same horizontal axis. (In unipolar mode, the axis is  $y = 0.5$ , and for bipolar mode it is the  $x$  axis). The *kShapeAmount* parameter specifies where

on the horizontal axis this point falls. When *kShapeAmount* is zero, the pivot point is in the middle of the input range, forming a straight line for the transfer function and thus causing no change in the input signal. As *kShapeAmount* changes from zero (0) to negative one (-1), the pivot point moves towards the left side of the graph, producing a phase distortion pattern like the Casio CZ's "sawtooth waveform". As it changes from zero (0) to positive one (1), the pivot point moves toward the right, producing an inverted pattern.

If the input to *pdhalf* is a phasor and the output is used to index a table, values for *kShapeAmount* that are less than zero will cause the first half of the table to be read more quickly than the second half. The reverse is true for values of *kShapeAmount* greater than zero. The rates at which the halves are read are calculated so that the frequency of the phasor is unchanged. Thus, this method of phase distortion can only produce higher partials in a harmonic series. It cannot produce inharmonic sidebands in the way that frequency modulation does.

*pdhalf* can work in either unipolar or bipolar modes. Unipolar mode is appropriate for signals like phasors that range between zero and some maximum value (selectable with *ifullscale*). Bipolar mode is appropriate for signals that range above and below zero by roughly equal amounts such as most audio signals. Applying *pdhalf* directly to an audio signal in this way results in a crude but adjustable sort of waveshaping/distortion.

A typical example of the use of *pdhalf* is

```

aphase  phasor  ifreq
apd      pdhalf  aphase, kamount
aout     tablei  apd, 1, 1

```

## Examples

Here is an example of the *pdhalf* opcode. It uses the file *pdhalf.csd* [examples/pdhalf.csd].

### Example 679. Example of the *pdhalf* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pdhalf.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 4

    idur          = p3
    iamp          = p4
    ifreq         = p5
    itable        = p6

    aenv          linseg    0, .001, 1.0, idur - .051, 1.0, .05, 0
    aosc          phasor    ifreq
    kamount       linseg    0.0, 0.02, -0.99, 0.05, -0.9, idur-0.06, 0.0
    apd           pdhalf    aosc, kamount

```

```
        aout      tablei      apd, itable, 1

                                outs      aenv*aout*iamp, aenv*aout*iamp

    endin

    </CsInstruments>
    <CsScore>
    f1 0 16385 10 1
    f2 0 16385 10 1 .5 .3333 .25 .5
    f3 0 16385 9 1 1 270          ; inverted cosine

    ; descending "just blues" scale

    ; pdhalf with cosine table
    ; (imitates the CZ-101 "sawtooth waveform")
    t 0 100
    i4 0 3 .6      512      3
    i. + . .      448
    i. + . .      384
    i. + . .      358.4
    i. + . .      341.33
    i. + . .      298.67
    i. + 5 .      256
    s
    ; pdhalf with a sine table
    t 0 120
    i4 0 3 .6      512      1
    i. + . .      448
    i. + . .      384
    i. + . .      358.4
    i. + . .      341.33
    i. + . .      298.67
    i. + 5 .      256
    s
    ; pdhalf with a sawtooth-like table
    t 0 150
    i4 0 3 .6      512      2
    i. + . .      448
    i. + . .      384
    i. + . .      358.4
    i. + . .      341.33
    i. + . .      298.67
    i. + 5 .      256
    e
    </CsScore>
    </CsoundSynthesizer>
```

## See Also

*pdhalfy*, *pdclip*

More information about phase distortion on Wikipedia: [http://en.wikipedia.org/wiki/Phase\\_distortion\\_synthesis](http://en.wikipedia.org/wiki/Phase_distortion_synthesis)

## Credits

Author: Anthony Kozar  
January 2008

New in Csound version 5.08

# pdhalfy

pdhalfy — Distorts a phasor for reading two unequal portions of a table in equal periods.

## Description

The *pdhalfy* opcode is a variation on the phase distortion synthesis method of the *pdhalf* opcode. It is useful for distorting a phasor in order to read two unequal portions of a table in the same number of samples.

## Syntax

```
aout pdhalfy ain, kShapeAmount [, ibipolar [, ifullscale]]
```

## Initialization

*ibipolar* -- an optional parameter specifying either unipolar (0) or bipolar (1) mode. Defaults to unipolar mode.

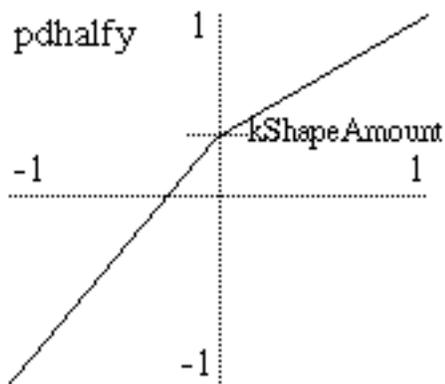
*ifullscale* -- an optional parameter specifying the range of input and output values. The maximum will be *ifullscale*. The minimum depends on the mode of operation: zero for unipolar or *-ifullscale* for bipolar. Defaults to 1.0 -- you should set this parameter to the maximum expected input value.

## Performance

*ain* -- the input signal to be distorted.

*aout* -- the output signal.

*kShapeAmount* -- the amount of distortion applied to the input. Must be between negative one and one (-1 to 1). An amount of zero means no distortion.



Transfer function created by *pdhalfy* and a negative *kShapeAmount*.

The *pdhalfy* opcode calculates a transfer function that is composed of two linear segments (see the graph). These segments meet at a "pivot point" which always lies on the same vertical axis. (In unipolar mode, the axis is  $x = 0.5$ , and for bipolar mode it is the  $y$  axis). So, *pdhalfy* is a variation of the *pdhalf* opcode that places the pivot point of the phase distortion pattern on a vertical axis instead of a horizontal axis.

The *kShapeAmount* parameter specifies where on the vertical axis this point falls. When *kShapeAmount* is zero, the pivot point is in the middle of the output range, forming a straight line for the transfer function and thus causing no change in the input signal. As *kShapeAmount* changes from zero (0) to negative one (-1), the pivot point downward towards the bottom of the graph. As it changes from zero (0) to positive one (1), the pivot point moves upward, producing an inverted pattern.

If the input to *pdhalfy* is a phasor and the output is used to index a table, the use of *pdhalfy* will divide the table into two segments of different sizes with each segment being mapped to half of the oscillator period. Values for *kShapeAmount* that are less than zero will cause less than half of the table to be read in the first half of the period of oscillation. The rest of the table will be read in the second half of the period. The reverse is true for values of *kShapeAmount* greater than zero. Note that the frequency of the phasor is always unchanged. Thus, this method of phase distortion can only produce higher partials in a harmonic series. It cannot produce inharmonic sidebands in the way that frequency modulation does. *pdhalfy* tends to have a milder quality to its distortion than *pdhalf*.

*pdhalfy* can work in either unipolar or bipolar modes. Unipolar mode is appropriate for signals like phasors that range between zero and some maximum value (selectable with *ifullscale*). Bipolar mode is appropriate for signals that range above and below zero by roughly equal amounts such as most audio signals. Applying *pdhalfy* directly to an audio signal in this way results in a crude but adjustable sort of waveshaping/distortion.

A typical example of the use of *pdhalfy* is

```

aphase  phasor    ifreq
apd     pdhalfy   aphase, kamount
aout    tablei    apd, 1, 1

```

## Examples

Here is an example of the *pdhalfy* opcode. It uses the file *pdhalfy.csd* [examples/pdhalfy.csd].

### Example 680. Example of the *pdhalfy* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pdhalfy.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 5

idur      = p3
iamp      = p4
ifreq     = p5
iamtinit  = p6      ; initial amount of phase distortion
iatt      = p7      ; attack time

```

```

isuslvl      = p8      ; sustain amplitude
idistdec     = p9      ; time for distortion amount to reach zero
itable       = p10

idec         = idistdec - iatt
irel         = .05
isus        = idur - (idistdec + irel)

aenv         linseg    0, iatt, 1.0, idec, isuslvl, isus, isuslvl, irel, 0, 0, 0
kamount      linseg    -iamtinit, idistdec, 0.0, idur-idistdec, 0.0
aosc         phasor    ifreq
apd          pdhalfy   aosc, kamount
aout         tablei    apd, itable, 1

outs         aenv*aout*iamp, aenv*aout*iamp

endin

</CsInstruments>
<CsScore>
f1 0 16385 10 1          ; sine
f3 0 16385 9 1 1 270    ; inverted cosine

; descending "just blues" scale

; pdhalfy with cosine table
t 0 100
i5 0 .333 .6 512 1.0 .02 0.5 .12 3
i. + . . 448 <
i. + . . 384 <
i. + . . 358.4 <
i. + . . 341.33 <
i. + . . 298.67 <
i. + 2 . 256 0.5
s

; pdhalfy with sine table
t 0 100
i5 0 .333 .6 512 1.0 .001 0.1 .07 1
i. + . . 448 <
i. + . . 384 <
i. + . . 358.4 <
i. + . . 341.33 <
i. + . . 298.67 <
i. + 2 . 256 0.5
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*pdhalf*, *pdclip*

More information about phase distortion on Wikipedia: [http://en.wikipedia.org/wiki/Phase\\_distortion\\_synthesis](http://en.wikipedia.org/wiki/Phase_distortion_synthesis)

## Credits

Author: Anthony Kozar  
January 2008

New in Csound version 5.08

# peak

peak — Maintains the output equal to the highest absolute value received.

## Description

These opcodes maintain the output k-rate variable as the peak absolute level so far received.

## Syntax

```
kres peak asig  
kres peak ksig
```

## Performance

*kres* -- Output equal to the highest absolute value received so far. This is effectively an input to the opcode as well, since it reads *kres* in order to decide whether to write something higher into it.

*ksig* -- k-rate input signal.

*asig* -- a-rate input signal.

## Examples

Here is an example of the peak opcode. It uses the file *peak.csd* [examples/peak.csd], and *beats.wav* [examples/beats.wav].

### Example 681. Example of the peak opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
; Audio out  Audio in  
-odac      -iadc      ;;RT audio I/O  
; For Non-realtime output leave only the line below:  
; -o peak.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>  
  
; Initialize the global variables.  
sr = 44100  
kr = 44100  
ksmps = 1  
nchnls = 1  
  
; Instrument #1 - play an audio file.  
instr 1  
; Capture the highest amplitude in the "beats.wav" file.  
asig soundin "beats.wav"  
kp peak asig
```

```
; Print out the peak value once per second.
printk 1, kp

out asig
endin

</CsInstruments>
<CsScore>

; Play Instrument #1, the audio file, for three seconds.
i 1 0 3
e

</CsScore>
</CsSoundSynthesizer>
```

Its output should include lines like this:

```
i 1 time 0.00002: 4835.00000
i 1 time 1.00002: 29312.00000
i 1 time 2.00002: 32767.00000
```

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

# pgmassign

pgmassign — Assigns an instrument number to a specified MIDI program.

## Description

Assigns an instrument number to a specified (or all) MIDI program(s).

By default, the instrument is the same as the program number. If the selected instrument is zero or negative or does not exist, the program change is ignored. This opcode is normally used in the orchestra header. Although, like *massign*, it also works in instruments.

## Syntax

```
pgmassign ipgm, inst[, ichn]
pgmassign ipgm, "insname"[, ichn]
```

## Initialization

*ipgm* -- MIDI program number (1 to 128). A value of zero selects all programs.

*inst* -- instrument number. If set to zero, or negative, MIDI program changes to *ipgm* are ignored. Currently, assignment to an instrument that does not exist has the same effect. This may be changed in a later release to print an error message.

*insname* -- A string (in double-quotes) representing a named instrument.

*ichn* (optional, defaults to zero) -- channel number. If zero, program changes are assigned on all channels.

You can disable the turning on of any instruments by using the following in the header:

```
massign 0, 0
pgmassign 0, 0
```

## Examples

Here is an example of the *pgmassign* opcode. It uses the file *pgmassign.csd* [examples/pgmassign.csd].

### Example 682. Example of the *pgmassign* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages  MIDI in
-odac      -iadc      -d          -M0   ;;RT audio I/O with MIDI in
; For Non-realtime ouput leave only the line below:
; -o pgmassign.wav -W ;; for file output any platform
```

```
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Program 55 (synth vox) uses Instrument #10.
pgmassign 55, 10

; Instrument #10.
instr 10
; Just an example, no working code in here!
endin

</CsInstruments>
<CsScore>

; Play Instrument #10 for one second.
i 10 0 1
e

</CsScore>
</CsoundSynthesizer>
```

Here is an example of the `pgmassign` opcode that will ignore program change events. It uses the file `pgmassign_ignore.csd` [examples/pgmassign\_ignore.csd].

### Example 683. Example of the `pgmassign` opcode that will ignore program change events.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages  MIDI in
-odac          -iadc      -d           -M0   ;;RT audio I/O with MIDI in
; For Non-realtime ouput leave only the line below:
; -o pgmassign_ignore.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Ignore all program change events.
pgmassign 0, -1

; Instrument #1.
instr 1
; Just an example, no working code in here!
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
```

```
i 1 0 1
e
```

```
</CsScore>
</CsoundSynthesizer>
```

Here is an advanced example of the `pgmassign` opcode. It uses the file `pgmassign_advanced.csd` [examples/pgmassign\_advanced.csd].

Don't forget that you must include the `-F` flag when using an external MIDI file like “pgmassign\_advanced.mid”.

### Example 684. An advanced example of the `pgmassign` opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -F pgmassign_advanced.mid ;;realtime audio out with midifile in
;-iadc ;;uncomment -iadc if real audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pgmassign_advanced.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

massign 1, 1 ; channels 1 to 4 use instr 1 by default
massign 2, 1
massign 3, 1
massign 4, 1

; pgmassign_advanced.mid can be found in /manual/examples
; pgmassign.mid has 4 notes with these parameters:
;
; Start time Channel Program
;
; note 1 0.5 1 10
; note 2 1.5 2 11
; note 3 2.5 3 12
; note 4 3.5 4 13

pgmassign 0, 0 ; disable program changes
pgmassign 11, 3 ; program 11 uses instr 3
pgmassign 12, 2 ; program 12 uses instr 2

; waveforms for instruments
itmp ftgen 1, 0, 1024, 10, 1
itmp ftgen 2, 0, 1024, 10, 1, 0.5, 0.3333, 0.25, 0.2, 0.1667, 0.1429, 0.125
itmp ftgen 3, 0, 1024, 10, 1, 0, 0.3333, 0, 0.2, 0, 0.1429, 0, 0.10101

instr 1 /* sine */

kcps cpsmidib 2 ; note frequency
asnd oscili .6, kcps, 1
outs asnd, asnd

endin

instr 2 /* band-limited sawtooth */
```

```
kcps cpsmidib 2 ; note frequency
asnd oscili .6, kcps, 2
outs asnd, asnd

endin

instr 3 /* band-limited square */

kcps cpsmidib 2 ; note frequency
asnd oscili .6, kcps, 3
outs asnd, asnd

endin

</CsInstruments>
<CsScore>

t 0 120
f 0 8.5 2 -2 0
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*midichn* and *massign*

## Credits

Author: Istvan Varga  
May 2002

New in version 4.20

# phaser1

phaser1 — First-order allpass filters arranged in a series.

## Description

An implementation of *iord* number of first-order allpass filters in series.

## Syntax

```
ares phaser1 asig, kfreq, kord, kfeedback [, iskip]
```

## Initialization

*iskip* (optional, default=0) -- used to control initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*kfreq* -- frequency (in Hz) of the filter(s). This is the frequency at which each filter in the series shifts its input by 90 degrees.

*kord* -- the number of allpass stages in series. These are first-order filters and can range from 1 to 4999.



### Note

Although *kord* is listed as k-rate, it is in fact accessed only at init-time. So if you are using a k-rate argument, it must be assigned with *init*.

*kfeedback* -- amount of the output which is fed back into the input of the allpass chain. With larger amounts of feedback, more prominent notches appear in the spectrum of the output. *kfeedback* must be between -1 and +1. for stability.

*phaser1* implements *iord* number of first-order allpass sections, serially connected, all sharing the same coefficient. Each allpass section can be represented by the following difference equation:

$$y(n) = C * x(n) + x(n-1) - C * y(n-1)$$

where  $x(n)$  is the input,  $x(n-1)$  is the previous input,  $y(n)$  is the output,  $y(n-1)$  is the previous output, and  $C$  is a coefficient which is calculated from the value of *kfreq*, using the bilinear z-transform.

By slowly varying *kfreq*, and mixing the output of the allpass chain with the input, the classic "phase shifter" effect is created, with notches moving up and down in frequency. This works best with *iord* between 4 and 16. When the input to the allpass chain is mixed with the output, 1 notch is generated for every 2 allpass stages, so that with *iord* = 6, there will be 3 notches in the output. With higher values for *iord*, modulating *kfreq* will result in a form of nonlinear pitch modulation.

## Examples

Here is an example of the phaser1 opcode. It uses the file *phaser1.csd* [examples/phaser1.csd].

### Example 685. Example of the phaser1 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc      -d          ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o phaser1.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; demonstration of phase shifting abilities of phaser1.
instr 1
; Input mixed with output of phaser1 to generate notches.
; Shows the effects of different iorder values on the sound
idur = p3
iamp = p4 * .05
iorder = p5 ; number of 1st-order stages in phaser1 network.
; Divide iorder by 2 to get the number of notches.
ifreq = p6 ; frequency of modulation of phaser1
ifeed = p7 ; amount of feedback for phaser1

kamp linseg 0, .2, iamp, idur - .2, iamp, .2, 0

iharms = (sr*.4) / 100

asig gbuzz 1, 100, iharms, 1, .95, 2 ; "Sawtooth" waveform modulation oscillator for phaser1 ugen.
kfreq oscili 5500, ifreq, 1
kmod = kfreq + 5600

aphs phaser1 asig, kmod, iorder, ifeed

out (asig + aphis) * iamp
endin

</CsInstruments>
<CsScore>

; inverted half-sine, used for modulating phaser1 frequency
f1 0 16384 9 .5 -1 0
; cosine wave for gbuzz
f2 0 8192 9 1 1 .25

; phaser1
i1 0 5 7000 4 .2 .9
i1 6 5 7000 6 .2 .9
i1 12 5 7000 8 .2 .9
i1 18 5 7000 16 .2 .9
i1 24 5 7000 32 .2 .9
i1 30 5 7000 64 .2 .9

```

e

```
</CsScore>  
</CsoundSynthesizer>
```

## Technical History

A general description of the differences between flanging and phasing can be found in Hartmann [1]. An early implementation of first-order allpass filters connected in series can be found in Beigel [2], where the bilinear z-transform is used for determining the phase shift frequency of each stage. Cronin [3] presents a similar implementation for a four-stage phase shifting network. Chamberlin [4] and Smith [5] both discuss using second-order allpass sections for greater control over notch depth, width, and frequency.

## References

1. Hartmann, W.M. "Flanging and Phasers." Journal of the Audio Engineering Society, Vol. 26, No. 6, pp. 439-443, June 1978.
2. Beigel, Michael I. "A Digital 'Phase Shifter' for Musical Applications, Using the Bell Labs (Alles-Fischer) Digital Filter Module." Journal of the Audio Engineering Society, Vol. 27, No. 9, pp. 673-676, September 1979.
3. Cronin, Dennis. "Examining Audio DSP Algorithms." Dr. Dobb's Journal, July 1994, p. 78-83.
4. Chamberlin, Hal. Musical Applications of Microprocessors. Second edition. Indianapolis, Indiana: Hayden Books, 1985.
5. Smith, Julius O. "An Allpass Approach to Digital Phasing and Flanging." Proceedings of the 1984 ICMC, p. 103-108.

## See Also

*phaser2*

Other information about phasers on Wikipedia: [http://en.wikipedia.org/wiki/Phaser\\_\(effect\)](http://en.wikipedia.org/wiki/Phaser_(effect))

## Credits

Author: Sean Costello  
Seattle, Washington  
1999

November 2002. Added a note about the *kord* parameter, thanks to Rasmus Ekman.

New in Csound version 4.0

# phaser2

phaser2 — Second-order allpass filters arranged in a series.

## Description

An implementation of *iord* number of second-order allpass filters in series.

## Syntax

```
ares phaser2 asig, kfreq, kq, kord, kmode, ksep, kfeedback
```

## Performance

*kfreq* -- frequency (in Hz) of the filter(s). This is the center frequency of the notch of the first allpass filter in the series. This frequency is used as the base frequency from which the frequencies of the other notches are derived.

*kq* -- Q of each notch. Higher Q values result in narrow notches. A Q between 0.5 and 1 results in the strongest "phasing" effect, but higher Q values can be used for special effects.

*kord* -- the number of allpass stages in series. These are second-order filters, and *iord* can range from 1 to 2499. With higher orders, the computation time increases.

*kfeedback* -- amount of the output which is fed back into the input of the allpass chain. With larger amounts of feedback, more prominent notches appear in the spectrum of the output. *kfeedback* must be between -1 and +1. for stability.

*kmode* -- used in calculation of notch frequencies.



### Note

Although *kord* and *kmode* are listed as k-rate, they are in fact accessed only at init-time. So if you are using k-rate arguments, they must be assigned with *init*.

*ksep* -- scaling factor used, in conjunction with *imode*, to determine the frequencies of the additional notches in the output spectrum.

*phaser2* implements *iord* number of second-order allpass sections, connected in series. The use of second-order allpass sections allows for the precise placement of the frequency, width, and depth of notches in the frequency spectrum. *iord* is used to directly determine the number of notches in the spectrum; e.g. for *iord* = 6, there will be 6 notches in the output spectrum.

There are two possible modes for determining the notch frequencies. When *imode* = 1, the notch frequencies are determined by the following function:

frequency of notch N =  $kbf + (ksep * kbf * N - 1)$

For example, with *imode* = 1 and *ksep* = 1, the notches will be in harmonic relationship with the notch frequency determined by *kfreq* (i.e. if there are 8 notches, with the first at 100 Hz, the next notches will be at 200, 300, 400, 500, 600, 700, and 800 Hz). This is useful for generating a "comb filtering" effect, with

the number of notches determined by *iord*. Different values of *ksep* allow for inharmonic notch frequencies and other special effects. *ksep* can be swept to create an expansion or contraction of the notch frequencies. A useful visual analogy for the effect of sweeping *ksep* would be the bellows of an accordion as it is being played - the notches will be separated, then compressed together, as *ksep* changes.

When *imode* = 2, the subsequent notches are powers of the input parameter *ksep* times the initial notch frequency specified by *kfreq*. This can be used to set the notch frequencies to octaves and other musical intervals. For example, the following lines will generate 8 notches in the output spectrum, with the notches spaced at octaves of *kfreq*:

```
aphs phaser2 ain, kfreq, 0.5, 8, 2, 2, 0
aout =      ain + aphis
```

When *imode* = 2, the value of *ksep* must be greater than 0. *ksep* can be swept to create a compression and expansion of notch frequencies (with more dramatic effects than when *imode* = 1).

## Examples

Here is an example of the phaser2 opcode. It uses the file *phaser2.csd* [examples/phaser2.csd].

### Example 686. Example of the phaser2 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o phaser2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 2          ; demonstration of phase shifting abilities of phaser2.
; Input mixed with output of phaser2 to generate notches.
; Demonstrates the interaction of imode and ksep.
idur = p3
iamp = p4 * .04
iorder = p5      ; number of 2nd-order stages in phaser2 network
ifreq = p6      ; not used
ifeed = p7      ; amount of feedback for phaser2
imode = p8      ; mode for frequency scaling
isep = p9      ; used with imode to determine notch frequencies
kamp linseg 0, .2, iamp, idur - .2, iamp, .2, 0
iharms = (sr*.4) / 100

; "Sawtooth" waveform exponentially decaying function, to control notch frequencies
asig gbuzz 1, 100, iharms, 1, .95, 2
kline expseg 1, idur, .005
aphs phaser2 asig, kline * 2000, .5, iorder, imode, isep, ifeed
```

```
    out (asig + aphs) * iamp
endin

</CsInstruments>
<CsScore>

; cosine wave for gbuzz
f2 0 8192 9 1 1 .25

; phaser2, imode=1
i2 00 10 7000 8 .2 .9 1 .33
i2 11 10 7000 8 .2 .9 1 2

; phaser2, imode=2
i2 22 10 7000 8 .2 .9 2 .33
i2 33 10 7000 8 .2 .9 2 2
e

</CsScore>
</CsoundSynthesizer>
```

## Technical History

A general description of the differences between flanging and phasing can be found in Hartmann [1]. An early implementation of first-order allpass filters connected in series can be found in Beigel [2], where the bilinear z-transform is used for determining the phase shift frequency of each stage. Cronin [3] presents a similar implementation for a four-stage phase shifting network. Chamberlin [4] and Smith [5] both discuss using second-order allpass sections for greater control over notch depth, width, and frequency.

## References

1. Hartmann, W.M. "Flanging and Phasers." Journal of the Audio Engineering Society, Vol. 26, No. 6, pp. 439-443, June 1978.
2. Beigel, Michael I. "A Digital 'Phase Shifter' for Musical Applications, Using the Bell Labs (Alles-Fischer) Digital Filter Module." Journal of the Audio Engineering Society, Vol. 27, No. 9, pp. 673-676, September 1979.
3. Cronin, Dennis. "Examining Audio DSP Algorithms." Dr. Dobb's Journal, July 1994, p. 78-83.
4. Chamberlin, Hal. Musical Applications of Microprocessors. Second edition. Indianapolis, Indiana: Hayden Books, 1985.
5. Smith, Julius O. "An Allpass Approach to Digital Phasing and Flanging." Proceedings of the 1984 ICMC, p. 103-108.

## See Also

*phaser1*

Other information about phasers on Wikipedia: [http://en.wikipedia.org/wiki/Phaser\\_\(effect\)](http://en.wikipedia.org/wiki/Phaser_(effect))

## Credits

Author: Sean Costello  
Seattle, Washington

1999

November 2002. Added a note about the *kord* and *kmode* parameters, thanks to Rasmus Ekman.

New in Csound version 4.0

# phasor

phasor — Produce a normalized moving phase value.

## Description

Produce a normalized moving phase value.

## Syntax

```
ares phasor xcps [ , iphs]
kres phasor kcps [ , iphs]
```

## Initialization

*iphs* (optional) -- initial phase, expressed as a fraction of a cycle (0 to 1). A negative value will cause phase initialization to be skipped. The default value is zero.

## Performance

An internal phase is successively accumulated in accordance with the *kcps* or *xcps* frequency to produce a moving phase value, normalized to lie in the range  $0 \leq \text{phs} < 1$ .

When used as the index to a *table* unit, this phase (multiplied by the desired function table length) will cause it to behave like an oscillator.

Note that *phasor* is a special kind of integrator, accumulating phase increments that represent frequency settings.

## Examples

Here is an example of the phasor opcode. It uses the file *phasor.csd* [examples/phasor.csd].

### Example 687. Example of the phasor opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if real audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o phasor.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
```

```
instr 1

ifn = 1 ;read table 1 with our index
ixmode = 1
kndx phasor p4
kfrq table kndx, ifn, ixmode
asig poscil .6, kfrq, 2 ;re-synthesize with sine
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 1025 -7 200 1024 2000 ;a line from 200 to 2,000
f 2 0 16384 10 1;sine wave

i 1 0 1 1 ;once per second
i 1 2 2 .5 ;once per 2 seconds
i 1 5 1 2 ;twice per second
e
</CsScore>
</CsoundSynthesizer>
```

## See also

The *Table Access* opcodes like: *table*, *tablei*, *table3* and *tab*.

Also: *phasorbnk*.

# phasorbnk

phasorbnk — Produce an arbitrary number of normalized moving phase values.

## Description

Produce an arbitrary number of normalized moving phase values, accessible by an index.

## Syntax

```
ares phasorbnk xcps, kndx, icnt [, iphs]
kres phasorbnk kcps, kndx, icnt [, iphs]
```

## Initialization

*icnt* -- maximum number of phasors to be used.

*iphs* -- initial phase, expressed as a fraction of a cycle (0 to 1). If -1 initialization is skipped. If *iphs*>1 each phasor will be initialized with a random value.

## Performance

*kndx* -- index value to access individual phasors

For each independent phasor, an internal phase is successively accumulated in accordance with the *kcps* or *xcps* frequency to produce a moving phase value, normalized to lie in the range  $0 \leq \text{phs} < 1$ . Each individual phasor is accessed by index *kndx*.

This phasor bank can be used inside a k-rate loop to generate multiple independent voices, or together with the *adsynt* opcode to change parameters in the tables used by *adsynt*.

## Examples

Here is an example of the phasorbnk opcode. It uses the file *phasorbnk.csd* [examples/phasorbnk.csd].

### Example 688. Example of the phasorbnk opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o phasorbnk.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
```

```

kr = 4410
ksmps = 10
nchnls = 1

; Generate a sinewave table.
giwave ftgen 1, 0, 1024, 10, 1

; Instrument #1
instr 1
; Generate 10 voices.
icnt = 10
; Empty the output buffer.
asum = 0
; Reset the loop index.
kindex = 0

; This loop is executed every k-cycle.
loop:
; Generate non-harmonic partials.
kcps = (kindex+1)*100+30
; Get the phase for each voice.
aphas phasorbk kcps, kindex, icnt
; Read the wave from the table.
asig table aphas, giwave, 1
; Accumulate the audio output.
asum = asum + asig

; Increment the index.
kindex = kindex + 1

; Perform the loop until the index (kindex) reaches
; the counter value (icnt).
if (kindex < icnt) kgoto loop

out asum*3000
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for two seconds.
i 1 0 2
e

</CsScore>
</CsoundSynthesizer>

```

Generate multiple voices with independent partials. This example is better with *adsynt*. See also the example under *adsynt*, for k-rate use of *phasorbk*.

## Credits

Author: Peter Neubäcker  
Munich, Germany  
August 1999

New in Csound version 3.58

# phs

phs — Obtains the phases of a complex-number array

## Description

This opcode returns the phases of a complex-number array, as a real-valued array with half the size of its input plus one. The reason for the extra point is to keep the array size equivalent to the output of the mags opcode (which is often used alongside this one).

## Syntax

```
kout[] phs kin[]
```

## Performance

*kout[]* -- output array containing the phases. It will be created if it does not exist.

*kin[]* -- input array containing the complex-valued real-imaginary input.

## Examples

Here is an example of the phs opcode. It uses the file *phs.csd* [examples/phs.csd].

### Example 689. Example of the phs opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-d -o dac
</CsOptions>
<CsInstruments>
/* ksmpls needs to be an integer div of
   hopsize */
ksmps = 64

instr 1

   ihopsize = 256 ; hopsize
   ifftsize = 2048 ; FFT size
   iolaps = ifftsize/ihopsize ; overlaps
   ibw = sr/ifftsize ; bin bandwidth
   kcnt init 0 ; counting vars
   krow init 0

   kOla[] init ifftsize ; overlap-add buffer
   kIn[] init ifftsize ; input buffer
   kOlph[] init ifftsize/2+1 ; old phases
   kPhs[] init ifftsize/2+1 ; synthesis phases
   kDeltaOut[] init ifftsize/2+1 ; synthesis freqs
   kMagsOut[] init ifftsize/2+1 ; synthesis mags
   kOut[][] init iolaps, ifftsize ; output buffers

   a1 diskin2 "fox.wav",1,0,1 ; audio input

   /* every hopsize samples */
```

```

if kcnt == ihopsize then
  /* window and take FFT */
  kWin[] window kIn,krow*ihopsize
  kSpec[] rfft kWin

  /* mags & phases */
  kMags[] mags kSpec
  kPha[] phs kSpec

  /* phase diffs */
  kDelta[] = kPha - kOlph
  kOlph = kPha

  /* pitch shift */
  ki = 0
  iscal = 1.25 ; maj 3rd up
  until ki == ifftsize/2 do
    if ki*iscal < ifftsize/2 then
      kDeltaOut[ki*iscal] = kDelta[ki]*iscal
      kMagsOut[ki*iscal] = kMags[ki]
    endif
    ki += 1
  od

  /* accum phases */
  kPhs = kDeltaOut + kPhs
  kSpec pol2rect kMagsOut,kPhs

  /* IFFT + window */
  kRow[] rfft kSpec
  kWin window kRow, krow*ihopsize
  /* place it on out buffer */
  kOut setrow kWin, krow

  /* zero the ola buffer */
  kOla = 0
  /* overlap-add */
  ki = 0
  until ki == iolaps do
    kRow getrow kOut, ki
    kOla = kOla + kRow
    ki += 1
  od

  /* update counters */
  krow = (krow+1)%iolaps
  kcnt = 0
endif

/* shift audio in/out of buffers */
kIn shiftin a1
a2 shiftout kOla
a1 delay a1, (ifftsize+ihopsize)/sr
out a1*0.3+a2/iolaps

/* increment counter */
kcnt += ksmpps

endin

</CsInstruments>
<CsScore>
i1 0 10
</CsScore>
</CsoundSynthesizer>

```

## See Also

Vectorial opcodes, array opcodes

## Credits

Author: Victor Lazzarini  
NUI Maynooth  
2014

New in version 6.04

# pindex

pindex — Returns the value of a specified pfield.

## Description

*pindex* returns the value of a specified pfield.

## Syntax

```
ivalue pindex ipfieldIndex
```

## Initialization

*ipfieldIndex* - pfield number to query.

*ivalue* - value of the pfield.

## Examples

Here is an example of the pindex opcode. It uses the file *pindex.csd* [examples/pindex.csd].

### Example 690. Example of the pindex opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages  MIDI in
-odac          -iadc    ; -d          -M0   ;;RT audio I/O with MIDI in
; For Non-realtime ouput leave only the line below:
;-o pindex.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
;Example by Anthony Kozar Dec 2006

instr 1
  inum    pcount
  index   init 1
  loop1:
    ivalue pindex index
    printf_i "p%d = %f\n", 1, index, ivalue
    index  = index + 1
    if (index <= inum) igoto loop1
  print inum
endin

</CsInstruments>
<CsScore>
i1 0 3 40 50          ; has 5 pfields
i1 1 2 80            ; has 5 due to carry
i1 2 1 40 50 60 70  ; has 7
e
```

```
</CsScore>
</CsoundSynthesizer>
```

The example will produce the following output:

```
new alloc for instr 1:
WARNING: instr 1 uses 3 p-fields but is given 5
p1 = 1.000000
p2 = 0.000000
p3 = 3.000000
p4 = 40.000000
p5 = 50.000000
instr 1: inum = 5.000
B 0.000 .. 1.000 T 1.000 TT 1.000 M: 0.0
new alloc for instr 1:
WARNING: instr 1 uses 3 p-fields but is given 5
p1 = 1.000000
p2 = 1.000000
p3 = 2.000000
p4 = 80.000000
p5 = 50.000000
instr 1: inum = 5.000
B 1.000 .. 2.000 T 2.000 TT 2.000 M: 0.0
new alloc for instr 1:
WARNING: instr 1 uses 3 p-fields but is given 7
p1 = 1.000000
p2 = 2.000000
p3 = 1.000000
p4 = 40.000000
p5 = 50.000000
p6 = 60.000000
p7 = 70.000000
instr 1: inum = 7.000
```

The warnings can be ignored, because the pfields are used indirectly through pindex instead of explicitly through p4, p5, etc.

Here is another example of the pindex opcode. It uses the file *pindex-2.csd* [examples/pindex-2.csd].

### Example 691. Second example of the pindex opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac ;;realtime audio
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pindex-2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

; This UDO returns a pfield value but returns 0 if it does not exist.
opcode mypvalue, i, i
index xin
inum pcount
if (index > inum) then
iout = 0.0
```

```

else
  iout pindex index
endif

xout iout
endop

; Envelope UDO that reads parameters from a flexible number of pfields
; Syntax:  kenv flexlinseg ipstart
;         ipstart is the first pfield of the envelope
;         parameters. Reads remaining pfields (up to 21 of them).
;         kenv is the output envelope.

opcode flexlinseg, k, i
  ipstart xin

  iep1 mypvalue ipstart
  iep2 mypvalue ipstart + 1
  iep3 mypvalue ipstart + 2
  iep4 mypvalue ipstart + 3
  iep5 mypvalue ipstart + 4
  iep6 mypvalue ipstart + 5
  iep7 mypvalue ipstart + 6
  iep8 mypvalue ipstart + 7
  iep9 mypvalue ipstart + 8
  iepa mypvalue ipstart + 9
  iepb mypvalue ipstart + 10
  iepc mypvalue ipstart + 11
  iepd mypvalue ipstart + 12
  iepe mypvalue ipstart + 13
  iepf mypvalue ipstart + 14
  iepg mypvalue ipstart + 15
  ieph mypvalue ipstart + 16
  iepi mypvalue ipstart + 17
  iepj mypvalue ipstart + 18
  iepk mypvalue ipstart + 19
  iepl mypvalue ipstart + 20

  kenv linseg iep1, iep2, iep3, iep4, iep5, iep6, iep7, iep8, \
        iep9, iepa, iepb, iepc, iepd, iepe, iepf, iepg, \
        ieph, iepi, iepj, iepk, iepl

  xout kenv
endop

instr 1
; This instrument only requires 3 pfields but can accept up to 24.
; (You will still get warnings about more than 3).
kenv flexlinseg 4 ; envelope params start at p4
aout oscili kenv*.5, 256, 1
  outs aout, aout

endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1

i1 0 2 0.0 1.0 1.0 1.0 0.0
i1 2 2 0.0 0.1 1.0 1.0 0.1 0.0
i1 4 2 0.0 0.5 0.0 ; one problem is that "missing" pfields carry
i1 6 2 0.0 0.5 0.0 ! ; now we can fix this problem with !
i1 8 10 0.0 3.0 1.0 0.3 0.1 0.3 1.0 0.3 0.1 0.3 1.0 0.3 0.1 0.8 0.9 5.0 0.0

e
</CsScore>
</CsoundSynthesizer>

```

The example will produce the following output:

```
WARNING: instr 1 uses 3 p-fields but is given 8
B 0.000 .. 2.000 T 2.000 TT 2.000 M: 0.49966 0.49966
WARNING: instr 1 uses 3 p-fields but is given 10
B 2.000 .. 4.000 T 4.000 TT 4.000 M: 0.50000 0.50000
WARNING: instr 1 uses 3 p-fields but is given 10
B 4.000 .. 6.000 T 6.000 TT 6.000 M: 0.49943 0.49943
WARNING: instr 1 uses 3 p-fields but is given 6
B 6.000 .. 8.000 T 8.000 TT 8.000 M: 0.00000 0.00000
WARNING: instr 1 uses 3 p-fields but is given 20
B 8.000 .. 18.000 T 18.000 TT 18.000 M: 0.49994 0.49994
```

## See Also

*pcount*

## Credits

Example by: Anthony Kozar

Dec. 2006

# pinker

pinker — Generates pink noise.

## Description

Generates pink noise (-3dB/oct response) by the *New Shade of Pink* algorithm of Stefan Stenzel.

## Syntax

```
ares pinker
```

## Performance

*pinker* generates pink noise (i.e., noise with equal energy in each octave), by the algorithm of Stefan Stenzel. For details of the algorithm look at <http://stenzel.waldorfmusic.de/post/pink/>.

## Examples

Here is an example of the pinker opcode. It uses the file *pinker.csd* [examples/pinker.csd].

### Example 692. Example of the pinker opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>

<CsInstruments>
ksmps = 12
nchnls = 1
0dbfs = 1

instr 1
  a1 pinker
  out a1
endin
</CsInstruments>

<CsScore>
i1 0 10
e
</CsScore>

</CsoundSynthesizer>
```

## Credits

Authors: Stefan Stenzel and John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2014

New in Csound Version 6.04

# pinkish

pinkish — Generates approximate pink noise.

## Description

Generates approximate pink noise (-3dB/oct response) by one of two different methods:

- a multirate noise generator after Moore, coded by Martin Gardner
- a filter bank designed by Paul Kellet

## Syntax

```
ares pinkish xin [, imethod] [, inumbands] [, iseed] [, iskip]
```

## Initialization

*imethod* (optional, default=0) -- selects filter method:

- 0 = Gardner method (default).
- 1 = Kellet filter bank.
- 2 = A somewhat faster filter bank by Kellet, with less accurate response.

*inumbands* (optional) -- only effective with Gardner method. The number of noise bands to generate. Maximum is 32, minimum is 4. Higher levels give smoother spectrum, but above 20 bands there will be almost DC-like slow fluctuations. Default value is 20.

*iseed* (optional, default=0) -- only effective with Gardner method. If non-zero, seeds the random generator. If zero, the generator will be seeded from current time. Default is 0.

*iskip* (optional, default=0) -- if non-zero, skip (re)initialization of internal state (useful for tied notes). Default is 0.

## Performance

*xin* -- for Gardner method: k- or a-rate amplitude. For Kellet filters: normally a-rate uniform random noise from rand (31-bit) or unrand, but can be any a-rate signal. The output peak value varies widely ( $\pm 15\%$ ) even over long runs, and will usually be well below the input amplitude. Peak values may also occasionally overshoot input amplitude or noise.

*pinkish* attempts to generate pink noise (i.e., noise with equal energy in each octave), by one of two different methods.

The first method, by Moore & Gardner, adds several (up to 32) signals of white noise, generated at octave rates (*sr*, *sr/2*, *sr/4* etc). It obtains pseudo-random values from an internal 32-bit generator. This random generator is local to each opcode instance and seedable (similar to *rand*).

The second method is a lowpass filter with a response approximating -3dB/oct. If the input is uniform white noise, it outputs pink noise. Any signal may be used as input for this method. The high quality filter is slower, but has less ripple and a slightly wider operating frequency range than less computationally intense versions. With the Kellet filters, seeding is not used.

The Gardner method output has some frequency response anomalies in the low-mid and high-mid frequency ranges. More low-frequency energy can be generated by increasing the number of bands. It is also a bit faster. The refined Kellert filter has very smooth spectrum, but a more limited effective range. The level increases slightly at the high end of the spectrum.

## Examples

Here is an example of the pinkish opcode. It uses the file *pinkish.csd* [examples/pinkish.csd].

### Example 693. Example of the pinkish opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc    -d      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o pinkish.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  awhite unirand 2.0

  ; Normalize to +/-1.0
  awhite = awhite - 1.0

  apink pinkish awhite, 1, 0, 0, 1

  out apink * 30000
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
e

</CsScore>
</CsoundSynthesizer>
```

Kellert-filtered noise for a tied note (*iskip* is non-zero).

## Credits

Authors: Phil Burk and John ffitc  
University of Bath/Codemist Ltd.  
Bath, UK

May 2000

New in Csound Version 4.07

Adapted for Csound by Rasmus Ekman

The noise bands method is due to F. R. Moore (or R. F. Voss), and was presented by Martin Gardner in an oft-cited article in Scientific American. The present version was coded by Phil Burk as the result of discussion on the music-dsp mailing list, with significant optimizations suggested by James McCartney.

The filter bank was designed by Paul Kellet, posted to the music-dsp mailing list.

The whole pink noise discussion was collected on a HTML page by Robin Whittle, which is currently available at <http://www.firstpr.com.au/dsp/pink-noise/>.

Added notes by Rasmus Ekman on September 2002.

# pitch

pitch — Tracks the pitch of a signal.

## Description

Using the same techniques as *spectrum* and *specptrk*, pitch tracks the pitch of the signal in octave point decimal form, and amplitude in dB.

## Syntax

```
koct, kamp pitch asig, iupdte, ilo, ihi, idbthresh [, ifrqs] [, iconf] \  
    [, istr] [, iocts] [, iq] [, inptls] [, irolloff] [, iskip]
```

## Initialization

*iupdte* -- length of period, in seconds, that outputs are updated

*ilo, ihi* -- range in which pitch is detected, expressed in octave point decimal

*idbthresh* -- amplitude, expressed in decibels, necessary for the pitch to be detected. Once started it continues until it is 6 dB down.

*ifrqs* (optional) -- number of divisions of an octave. Default is 12 and is limited to 120.

*iconf* (optional) -- the number of conformations needed for an octave jump. Default is 10.

*istr* (optional) -- starting pitch for tracker. Default value is  $(ilo + ihi)/2$ .

*iocts* (optional) -- number of octave decimations in spectrum. Default is 6.

*iq* (optional) -- Q of analysis filters. Default is 10.

*inptls* (optional) -- number of harmonics, used in matching. Computation time increases with the number of harmonics. Default is 4.

*iroloff* (optional) -- amplitude rolloff for the set of filters expressed as fraction per octave. Values must be positive. Default is 0.6.

*iskip* (optional) -- if non-zero, skips initialization. Default is 0.

## Performance

*koct* -- The pitch output, given in the octave point decimal format.

*kamp* -- The amplitude output.

*pitch* analyzes the input signal, *asig*, to give a pitch/amplitude pair of outputs, for the strongest frequency in the signal. The value is updated every *iupdte* seconds.

The number of partials and rolloff fraction can effect the pitch tracking, so some experimentation may be necessary. Suggested values are 4 or 5 harmonics, with rolloff 0.6, up to 10 or 12 harmonics with rolloff 0.75 for complex timbres, with a weak fundamental.

## Examples

Here is an example of the pitch opcode. It uses the file *pitch.csd* [examples/pitch.csd].

### Example 694. Example of the pitch opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if real audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pitch.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;clean audio
asig soundin p4
  outs asig, asig
endin

instr 2 ;use pitch

iupdte = 0.001 ;high definition
ilo = 6
ihi = 10
idbthresh = 10
ifrgs = 12
iconf = 10
istrt = 8

asig soundin p4
koct, kamp pitch asig, iupdte, ilo, ihi, idbthresh, ifrgs, iconf, istrt
kamp = kamp*.00005 ;lower volume
kcps = cpsoct(koct)
asig poscil kamp, kcps, 1 ;re-synthesize with sawtooth
  outs asig, asig

endin
</CsInstruments>
<CsScore>
f1 0 16384 10 1 0.5 0.3 0.25 0.2 0.167 0.14 0.125 .111 ;sawtooth

i 1 0 3 "fox.wav"
i 2 3 3 "fox.wav"
i 1 6 3 "mary.wav"
i 2 9 3 "mary.wav"
i 1 12 3 "beats.wav"
i 2 15 3 "beats.wav"
e
</CsScore>
</CsoundSynthesizer>

```

## Credits

Author: John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK  
April 1999

New in Csound version 3.54

# pitchamdf

pitchamdf — Follows the pitch of a signal based on the AMDF method.

## Description

Follows the pitch of a signal based on the AMDF method (Average Magnitude Difference Function). Outputs pitch and amplitude tracking signals. The method is quite fast and should run in realtime. This technique usually works best for monophonic signals.

## Syntax

```
kcps, krms pitchamdf asig, imincps, imaxcps [, icps] [, imedi] \  
[, idowns] [, iexcps] [, irmsmedi]
```

## Initialization

*imincps* -- estimated minimum frequency (expressed in Hz) present in the signal

*imaxcps* -- estimated maximum frequency present in the signal

*icps* (optional, default=0) -- estimated initial frequency of the signal. If 0,  $icps = (imincps + imaxcps) / 2$ . The default is 0.

*imedi* (optional, default=1) -- size of median filter applied to the output *kcps*. The size of the filter will be  $imedi * 2 + 1$ . If 0, no median filtering will be applied. The default is 1.

*idowns* (optional, default=1) -- downsampling factor for *asig*. Must be an integer. A factor of *idowns* > 1 results in faster performance, but may result in worse pitch detection. Useful range is 1 - 4. The default is 1.

*iexcps* (optional, default=0) -- how frequently pitch analysis is executed, expressed in Hz. If 0, *iexcps* is set to *imincps*. This is usually reasonable, but experimentation with other values may lead to better results. Default is 0.

*irmsmedi* (optional, default=0) -- size of median filter applied to the output *krms*. The size of the filter will be  $irmsmedi * 2 + 1$ . If 0, no median filtering will be applied. The default is 0.

## Performance

*kcps* -- pitch tracking output

*krms* -- amplitude tracking output

*pitchamdf* usually works best for monophonic signals, and is quite reliable if appropriate initial values are chosen. Setting *imincps* and *imaxcps* as narrow as possible to the range of the signal's pitch, results in better detection and performance.

Because this process can only detect pitch after an initial delay, setting *icps* close to the signal's real initial pitch prevents spurious data at the beginning.

The median filter prevents *kcps* from jumping. Experiment to determine the optimum value for *imedi* for a given signal.

Other initial values can usually be left at the default settings. Lowpass filtering of *asig* before passing it to *pitchamdf*, can improve performance, especially with complex waveforms.

## Examples

Here is an example of the *pitchamdf* opcode. It uses the file *pitchamdf.csd* [examples/pitchamdf.csd].

### Example 695. Example of the *pitchamdf* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if real audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pitchamdf.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;clean audio

asig soundin p4
outs asig, asig
endin

instr 2 ;use pitch

asig soundin p4
asig tone asig, 1000 ;lowpass-filter
kcps, krms pitchamdf asig, 100, 500, 200
asig poscil krms, kcps, 1 ;re-synthesize with sawtooth
outs asig, asig

endin
</CsInstruments>
<CsScore>
f1 0 16384 10 1 0.5 0.3 0.25 0.2 0.167 0.14 0.125 .111 ;sawtooth

i 1 0 3 "fox.wav"
i 2 3 3 "fox.wav"
i 1 6 3 "mary.wav"
i 2 9 3 "mary.wav"
i 1 12 3 "beats.wav"
i 2 15 3 "beats.wav"
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Peter Neubäcker  
Munich, Germany  
August 1999

New in Csound version 3.59

# planet

planet — Simulates a planet orbiting in a binary star system.

## Description

*planet* simulates a planet orbiting in a binary star system. The outputs are the x, y and z coordinates of the orbiting planet. It is possible for the planet to achieve escape velocity by a close encounter with a star. This makes this system somewhat unstable.

## Syntax

```
ax, ay, az planet kmass1, kmass2, ksep, ix, iy, iz, ivx, ivy, ivz, idelta \  
[, ifriction] [, iskip]
```

## Initialization

*ix, iy, iz* -- the initial x, y and z coordinates of the planet

*ivx, ivy, ivz* -- the initial velocity vector components for the planet.

*idelta* -- the step size used to approximate the differential equation.

*ifriction* (optional, default=0) -- a value for friction, which can be used to keep the system from blowing up

*iskip* (optional, default=0) -- if non zero skip the initialisation of the filter. (New in Csound version 4.23f13 and 5.0)

## Performance

*ax, ay, az* -- the output x, y, and z coordinates of the planet

*kmass1* -- the mass of the first star

*kmass2* -- the mass of the second star

## Examples

Here is an example of the planet opcode. It uses the file *planet.csd* [examples/planet.csd].

### Example 696. Example of the planet opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
; Audio out   Audio in   No messages  
-odac        -iadc       -d           ;;RT audio I/O  
; For Non-realtime output leave only the line below:  
; -o planet.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>  
  
; Initialize the global variables.
```

```
sr = 44100
kr = 44100
ksmps = 1
nchnls = 2

; Instrument #1 - a planet orbiting in 3D space.
instr 1
; Create a basic tone.
kamp init 5000
kcps init 440
ifn = 1
asnd oscil kamp, kcps, ifn

; Figure out its X, Y, Z coordinates.
km1 init 0.5
km2 init 0.35
ksep init 2.2
ix = 0
iy = 0.1
iz = 0
ivx = 0.5
ivy = 0
ivz = 0
ih = 0.0003
ifric = -0.1
ax1, ay1, az1 planet km1, km2, ksep, ix, iy, iz, \
                    ivx, ivy, ivz, ih, ifric

; Place the basic tone within 3D space.
kx downsamp ax1
ky downsamp ay1
kz downsamp az1
idist = 1
ift = 0
imode = 1
imdel = 1.018853416
iovr = 2
aw2, ax2, ay2, az2 spat3d asnd, kx, ky, kz, idist, \
                        ift, imode, imdel, iovr

; Convert the 3D sound to stereo.
aleft = aw2 + ay2
aright = aw2 - ay2

outs aleft, aright
endin

</CsInstruments>
<CsScore>

; Table #1 a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for 10 seconds.
i 1 0 10
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

More information on this opcode: <http://www.csoundjournal.com/issue9/FlutesInOrbit.html> , written by Brian Redfern.

## Credits

Author: Hans Mikelson  
December 1998

New in Csound version 3.50

# platerrev

platerrev — Models the reverberation of a metal plate.

## Description

Models the reverberation of a rectangular metal plate with settable physical characteristics when excited by audio signal(s).

## Syntax

```
a1[, a2, ...] platerrev itabexcite. itabouts, kbndry, iaspect, istiff, idecay, iloss, aexcitel[, aexcite
```

## Initialization

*itabexcite* -- number of a table that contains triplets for each excitation signal, (frequency, radius, initial phase in radians). The radius should be less than 1. These control where the excitation happens. The values in the table for frequency and radius may be changed in performance, with the proviso about clicking if the changes are too large.

*itabouts* -- number of a table that contains triplets for each output signal, (frequency, radius, initial phase in radians). See *itabexcite* description.

*kbndry* -- boundary condition of the plate; 0 = free, 1 = clamped, 2 = pivoting. Other values are undefined. Thus parameter may be changed at k-rate but that might give rise to clicks.

*iaspect* -- plate aspect ratio which should be less than or equal to 1.

*istiff* -- plate stiffness parameter (set around 1 or less for plate reverb).

*idecay* -- time taken for 30 dB decay

*iloss* -- loss parameter for high-frequency damping (value about 0.001 suggested).

## Performance

A metal plate is excited by a stereo signal and the resulting reverberation captured.

*aexciten* -- excitation signal to be inserted into the plate.

## Examples

Here is an example of the *platerrev* opcode. It uses the file *plate.csd* [examples/plate.csd].

### Example 697. Example of the *platerrev* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>
```

```
; Select audio/midi flags here according to platform
;-odac    ;;realtime audio out
;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
-o plate.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
; Note: this example is heavy on CPU
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ab    diskin2 "beats.wav", 1
al,ar platerrev 1, 2, p4, 0.73, 1.0, 5.0, 0.001, ab,ab
outs    al*.25,ar*.25

endin
</CsInstruments>
<CsScore>
f1 0 8 -2 0.3 0.3875 0.39274 0.32 0.85714 0.78548
f2 0 8 -2 0.2 0.666667 1.57097 0.24 0.75 0.78548
i1 0 4 1
i1 + 4 2
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Stefan Bilbao  
University of Edinburgh, UK  
Author: John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 5.17.12

# plltrack

plltrack — Tracks the pitch of a signal.

## Description

*plltrack*, a pitch tracker based on a phase-locked loop algorithm, described in Zolzer, U, Sankarababu, S.V. and Moller, S, "PLL-based Pitch Detection and Tracking for Audio Signals. Proc. of IHH-MSP 2012".

## Syntax

```
acps, alock plltrack asig, kd [, kloopf, kloopq, klf, khf, kthresh]
```

## Performance

*acps* -- estimated pitch in Hz.

*alock* -- phase lock indicator, a phase error indicating the quality of the tracking, with values between 0 and 1. Higher values indicate good tracking

*kd* -- PLL feedback gain, controls frequency range of PLL (between 0 and 1). Higher values increase the range of the tracking.

*kloopf* -- PLL LP filter cf, controls frequency range of PLL (opt, defaults to 20Hz).

*kloopq* -- PLL LP filter Q, controls rise time of FO step (opt, defaults to 1/3)

*klf* -- lowest tracking freq (opt, defaults to 20Hz)

*khf* -- highest tracking freq (opt, defaults to 1500Hz)

*kthresh* -- tracking signal level threshold (optional, defaults to 0.001, equiv to -60dBfs)

*plltrack* analyzes the input signal, *asig*, estimating the fundamental of a monophonic signal. Its output is updated every sample.

## Examples

Here is an example of the *plltrack* operator. It uses the file *plltrack.csd* [examples/plltrack.csd].

### Example 698. Example of the *plltrack* operator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o plltrack.wav -W ;; for file output any platform
</CsOptions>
```

```
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
Odbfs = 1

instr 1

kd = p4
a1 diskin2 "fox.wav", 1,0,1
apitch, aloc plltrack a1, kd
krms rms a1
krms port krms, 0.01
asig buzz krms, apitch, 10, 1
      outs asig, asig ;mix in some dry signal as well

endin
</CsInstruments>
<CsScore>
f1 0 65536 10 1 ;sine wave

i 1 0 6 0.1
i 1 7 6 0.3 ;more feedback

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
NUI, Maynooth.  
Maynooth, Ireland  
September, 2012

New in Csound version 5.18.031

# pluck

pluck — Produces a naturally decaying plucked string or drum sound.

## Description

Audio output is a naturally decaying plucked string or drum sound based on the Karplus-Strong algorithms.

## Syntax

```
ares pluck kamp, kcps, icps, ifn, imeth [, iparm1] [, iparm2]
```

## Initialization

*icps* -- intended pitch value in Hz, used to set up a buffer of 1 cycle of audio samples which will be smoothed over time by a chosen decay method. *icps* normally anticipates the value of *kcps*, but may be set artificially high or low to influence the size of the sample buffer.

*ifn* -- table number of a stored function used to initialize the cyclic decay buffer. If *ifn* = 0, a random sequence will be used instead.

*imeth* -- method of natural decay. There are six, some of which use parameter values that follow.

1. simple averaging. A simple smoothing process, uninfluenced by parameter values.
2. stretched averaging. As above, with smoothing time stretched by a factor of *iparm1* ( $\geq 1$ ).
3. simple drum. The range from pitch to noise is controlled by a 'roughness factor' in *iparm1* (0 to 1). Zero gives the plucked string effect, while 1 reverses the polarity of every sample (octave down, odd harmonics). The setting .5 gives an optimum snare drum.
4. stretched drum. Combines both roughness and stretch factors. *iparm1* is roughness (0 to 1), and *iparm2* the stretch factor ( $\geq 1$ ).
5. weighted averaging. As method 1, with *iparm1* weighting the current sample (the status quo) and *iparm2* weighting the previous adjacent one.  $iparm1 + iparm2$  must be  $\leq 1$ .
6. 1st order recursive filter, with coefs .5. Unaffected by parameter values.

*iparm1*, *iparm2* (optional) -- parameter values for use by the smoothing algorithms (above). The default values are both 0.

## Performance

*kamp* -- the output amplitude.

*kcps* -- the resampling frequency in cycles-per-second.

An internal audio buffer, filled at i-time according to *ifn*, is continually resampled with periodicity *kcps* and the resulting output is multiplied by *kamp*. Parallel with the sampling, the buffer is smoothed to simulate the effect of natural decay.

Plucked strings (1, 2, 5, 6) are best realized by starting with a random noise source, which is rich in initial harmonics. Drum sounds (methods 3, 4) work best with a flat source (wide pulse), which produces a deep noise attack and sharp decay.

The original Karplus-Strong algorithm used a fixed number of samples per cycle, which caused serious quantization of the pitches available and their intonation. This implementation resamples a buffer at the exact pitch given by *kcps*, which can be varied for vibrato and glissando effects. For low values of the orch sampling rate (e.g. *sr* = 10000), high frequencies will store only very few samples (*sr* / *icps*). Since this may cause noticeable noise in the resampling process, the internal buffer has a minimum size of 64 samples. This can be further enlarged by setting *icps* to some artificially lower pitch.

## Examples

Here is an example of the pluck opcode. It uses the file *pluck.csd* [examples/pluck.csd].

### Example 699. Example of the pluck opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if real audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pluck.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kcps = 220
icps = 220
ifn = 0
imeth = p4

asig pluck 0.7, 220, 220, ifn, imeth, .1, 10
outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 5 1
i 1 5 5 4 ;needs 2 extra parameters (iparm1, iparm2)
i 1 10 5 6
e
</CsScore>
</CsoundSynthesizer>
```

# poisson

`poisson` — Poisson distribution random number generator (positive values only).

## Description

Poisson distribution random number generator (positive values only). This is an x-class noise generator.

## Syntax

```
ares poisson klambda
```

```
ires poisson klambda
```

```
kres poisson klambda
```

## Performance

*ares*, *kres*, *ires* - number of events occurring (always an integer).

*klambda* - the expected number of occurrences that occur during the rate interval.

## Adapted from Wikipedia:

In probability theory and statistics, the Poisson distribution is a discrete probability distribution. It expresses the probability of a number of events occurring in a fixed period of time if these events occur with a known average rate, and are independent of the time since the last event.

The Poisson distribution describing the probability that there are exactly  $k$  occurrences ( $k$  being a non-negative integer,  $k = 0, 1, 2, \dots$ ) is:

$$f(k; \lambda) = \frac{e^{-\lambda} \lambda^k}{k!},$$

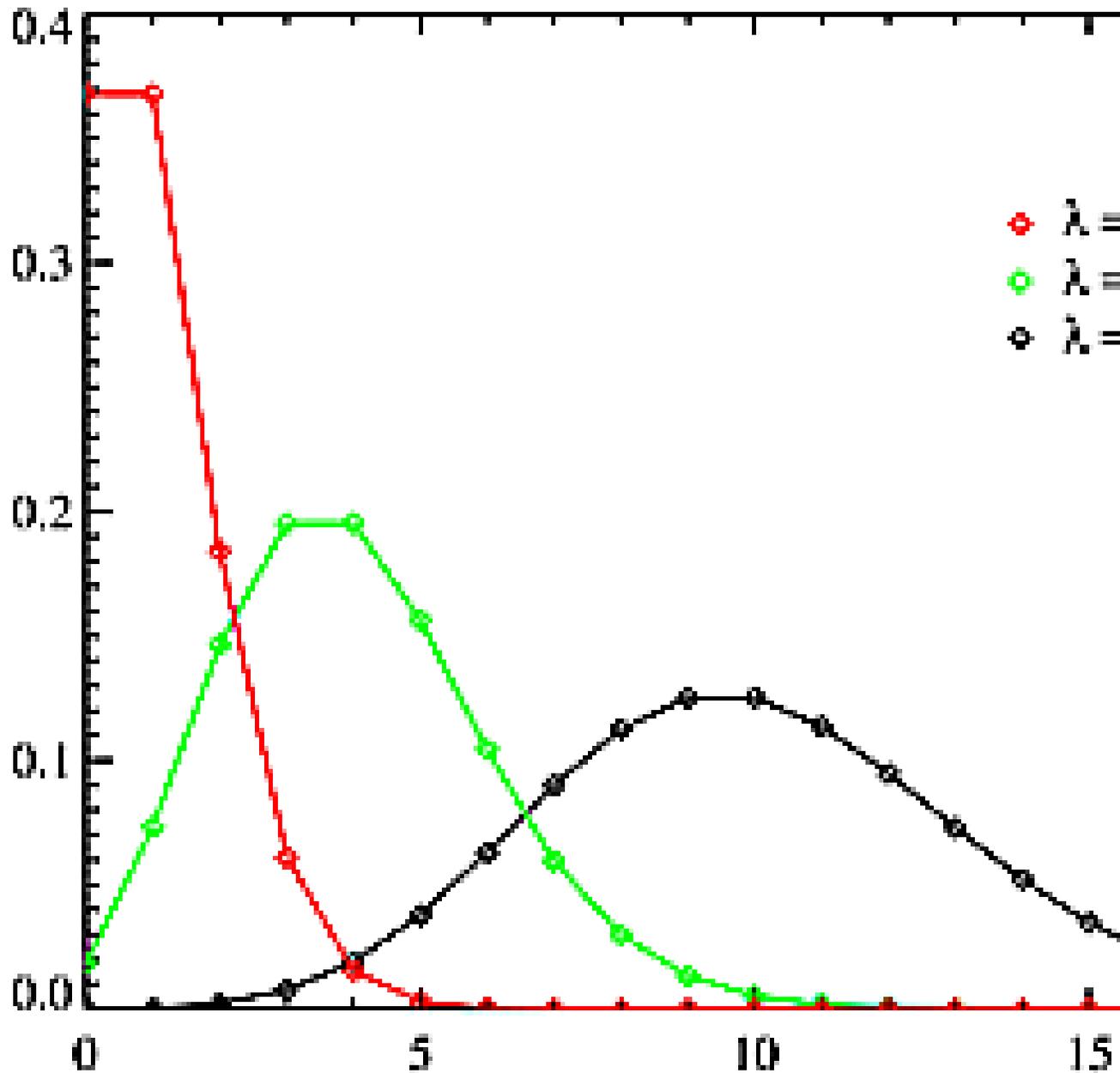
where:

- $\lambda$  is a positive real number, equal to the expected number of occurrences that occur during the given interval. For instance, if the events occur on average every 4 minutes, and you are interested in the number of events occurring in a 10 minute interval, you would use as model a Poisson distribution with  $\lambda = 10/4 = 2.5$ . This parameter is called *klambda* on the *poisson* opcodes.
- $k$  refers to the number of i-, k- or a- periods elapsed.

The Poisson distribution arises in connection with Poisson processes. It applies to various phenomena of discrete nature (that is, those that may happen 0, 1, 2, 3, ... times during a given period of time or in a given area) whenever the probability of the phenomenon happening is constant in time or space. Examples of events that can be modelled as Poisson distributions include:

- The number of cars that pass through a certain point on a road (sufficiently distant from traffic lights) during a given period of time.

- The number of spelling mistakes one makes while typing a single page.
- The number of phone calls at a call center per minute.
- The number of times a web server is accessed per minute.
- The number of roadkill (animals killed) found per unit length of road.
- The number of mutations in a given stretch of DNA after a certain amount of radiation.
- The number of unstable nuclei that decayed within a given period of time in a piece of radioactive substance. The radioactivity of the substance will weaken with time, so the total time interval used in the model should be significantly less than the mean lifetime of the substance.
- The number of pine trees per unit area of mixed forest.
- The number of stars in a given volume of space.
- The distribution of visual receptor cells in the retina of the human eye.
- The number of viruses that can infect a cell in cell culture.



A diagram showing the Poisson distribution.

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the poisson opcode. It uses the file *poisson.csd* [examples/poisson.csd]. It is written for \*NIX systems, and will generate errors on Windows.

## Example 700. Example of the poisson opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      -iadc      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o poisson.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
ksmps = 441 ;ksmps set deliberately high to have few k-periods per second
nchnls = 1

; Instrument #1.
instr 1
; Generates a random number in a poisson distribution.
; klambda = 1

i1 poisson 1

print i1
endin

instr 2

kres poisson p4
printk (ksmps/sr),kres ;prints every k-period
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
i 2 1 0.2 0.5
i 2 2 0.2 4 ;average 4 events per k-period
i 2 3 0.2 20 ;average 20 events per k-period
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*seed, betarand, bexpnd, cauchy, exprand, gauss, linrand, pcauchy, trirand, unirand, weibull*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

# pol2rect

pol2rect — Polar to rectangular format conversion.

## Description

Converts an input array in magnitude-phase format to real-imaginary format.

## Syntax

```
kout[] pol2rect kin[]  
kout[] pol2rect kmags[], kphs[]
```

## Performance

*kout[]* -- output array containing the complex-valued real-imaginary output. It will be created if it does not exist.

*kin[]* -- input array containing the complex-valued magnitude-phase input.

*kmags[]* -- input array containing the real-valued magnitude input.

*kphs[]* -- input array containing the real-valued phase input.

## Examples

Here is an example of the pol2rect opcode. It uses the file *pol2rect.csd* [examples/pol2rect.csd].

### Example 701. Example of the pol2rect opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
-d -o dac  
</CsOptions>  
<CsInstruments>  
/* ksmpls needs to be an integer div of  
   hopsize */  
ksmps = 64  
  
instr 1  
  
   ihopsize = 256 ; hopsize  
   ifftsize = 1024 ; FFT size  
   iolaps = ifftsize/ihopsize ; overlaps  
   ibw = sr/ifftsize ; bin bandwidth  
   kcmt init 0 ; counting vars  
   krow init 0  
  
   kOla[] init ifftsize ; overlap-add buffer  
   kIn[] init ifftsize ; input buffer  
   kOut[][] init iolaps, ifftsize ; output buffers
```

```

a1 diskin2 "fox.wav",1,0,1 ; audio input

/* every hopsize samples */
if kcnt == ihopsize then
  /* window and take FFT */
  kWin[] window kIn,krow*ihopsize
  kSpec[] rfft kWin

  kSpec rect2pol kSpec

  /* reduce mags between high and low freqs */
  ilow = 0
  ihigh = 1000
  ki = int(ilow/ibw)
  until ki >= int(ihigh/ibw) do
    kSpec[ki] = kSpec[ki]*0.1
    ki += 2
  od

  kSpec pol2rect kSpec

  /* IFFT + window */
  kRow[] rifft kSpec
  kWin window kRow, krow*ihopsize
  /* place it on out buffer */
  kOut setrow kWin, krow

  /* zero the ola buffer */
  kOla = 0
  /* overlap-add */
  ki = 0
  until ki == iolaps do
    kRow getrow kOut, ki
    kOla = kOla + kRow
    ki += 1
  od

  /* update counters */
  krow = (krow+1)%iolaps
  kcnt = 0
endif

/* shift audio in/out of buffers */
kIn shiftin a1
a2 shiftout kOla
  out a2/iolaps

/* increment counter */
kcnt += ksmps

endin

</CsInstruments>
<CsScore>
i1 0 10
</CsScore>
</CsoundSynthesizer>

```

## See Also

Vectorial opcodes, array opcodes

## Credits

Author: Victor Lazzarini

NUI Maynooth  
2014

New in version 6.04

# polyaft

`polyaft` — Returns the polyphonic after-touch pressure of the selected note number.

## Description

`polyaft` returns the polyphonic pressure of the selected note number, optionally mapped to a user-specified range.

## Syntax

```
ires polyaft inote [, ilow] [, ihigh]
kres polyaft inote [, ilow] [, ihigh]
```

## Initialization

`inote` -- note number. Normally set to the value returned by `notnum`

`ilow` (optional, default: 0) -- lowest output value

`ihigh` (optional, default: 127) -- highest output value

## Performance

`kres` -- polyphonic pressure (aftertouch).

## Examples

Here is an example of the `polyaft` opcode. It uses the file `polyaft.csd` [examples/polyaft.csd].

Don't forget that you must include the `-F` flag when using an external MIDI file like “polyaft.mid”.

### Example 702. Example of the `polyaft` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages  MIDI in
-odac        -iadc      -d           -M0   ;;RT audio I/O with MIDI in
; For Non-realtime ouput leave only the line below:
; -o polyaft.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 10
nchnls = 1

massign 1, 1
itmp ftgen 1, 0, 1024, 10, 1 ; sine wave
```

```
instr 1

kcps cpsmidib 2 ; note frequency
inote notnum ; note number
kaft polyaft inote, 0, 127 ; aftertouch
; interpolate aftertouch to eliminate clicks
ktmp phasor 40
ktmp trigger 1 - ktmp, 0.5, 0
kaft tlineto kaft, 0.025, ktmp
; map to sine curve for crossfade
kaft = sin(kaft * 3.14159 / 254) * 22000

asnd oscili kaft, kcps, 1

out asnd

endin

</CsInstruments>
<CsScore>

t 0 120
f 0 9 2 -2 0
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Added thanks to an email from Istvan Varga

New in version 4.12

# polynomial

polynomial — Efficiently evaluates a polynomial of arbitrary order.

## Description

The *polynomial* opcode calculates a polynomial with a single a-rate input variable. The polynomial is a sum of any number of terms in the form  $kn*x^n$  where  $kn$  is the  $n$ th coefficient of the expression. These coefficients are k-rate values.

## Syntax

```
out polynomial ain, k0 [, k1 [, k2 [...]]]
```

## Performance

*ain* -- the input signal used as the independent variable of the polynomial ("x").

*out* -- the output signal ("y").

*k0, k1, k2, ...* -- the coefficients for each term of the polynomial.

If we consider the input parameter *ain* to be "x" and the output *out* to be "y", then the *polynomial* opcode calculates the following equation:

$$y = k_0 + k_1*x + k_2*x^2 + k_3*x^3 + \dots$$

## Examples

Here is an example of the polynomial opcode. It uses the file *polynomial.csd* [examples/polynomial.csd].

### Example 703. Example of the polynomial opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o polynomial.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

; The polynomial y=x^n where n is odd always produces a curve
; that traverses the range [-1, 1] when the input is within
; the same range. Therefore, we can use one of these curves
; to make a nonlinear phasor that repeatedly reads a table
```

```

; from beginning to end like a linear phasor (maintaining
; continuity) but that distorts the waveform in the table.

instr 4 ; This instrument demonstrates phase distortion with x^3

idur = p3
iamp = p4
ifreq = p5
itable = p6

aenv linseg 0, .001, 1.0, idur - .051, 1.0, .05, 0 ; declicking envelope
aosc phasor ifreq ; create a linear phasor
apd polynomial aosc, 0, 0, 0, 1 ; distort the phasor with x^3
aout tablei apd, itable, 1 ; read a sine table with the nonlinear phasor
outs aenv*aout*iamp, aenv*aout*iamp

endin

instr 5 ; This instrument demonstrates phase distortion with x^11

idur = p3
iamp = p4
ifreq = p5
itable = p6

aenv linseg 0, .001, 1.0, idur - .051, 1.0, .05, 0 ; declicking envelope
aosc phasor ifreq ; create a linear phasor
apd polynomial aosc, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 ; distort the phasor with x^11
aout tablei apd, itable, 1 ; read a sine table with the nonlinear phasor
outs aenv*aout*iamp, aenv*aout*iamp

endin

instr 6 ; This instrument crossfades between a pure sine and one distorted with x^11

idur = p3
iamp = p4
ifreq = p5
itable = p6

aenv linseg 0, .001, 1.0, idur - .051, 1.0, .05, 0 ; declicking envelope
aosc phasor ifreq ; create a linear phasor
aout3 tablei aosc, itable, 1 ; read a sine table without the linear phasor
apd11 polynomial aosc, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 ; distort the phasor with x^11
aout11 tablei apd11, itable, 1 ; read a sine table with the nonlinear phasor
kamount linseg 1.0, 0.05, 0.9, 1.0, 0.0 ; crossfade between two outputs
aout = aout3*kamount + aout11*(1.0 - kamount)
outs aenv*aout*iamp, aenv*aout*iamp

endin
</CsInstruments>
<CsScore>
f1 0 16385 10 1 ; sine wave

; descending "just blues" scale

t 0 100
i4 0 .333 .7 512 1
i. + . . 448
i. + . . 384
i. + . . 360
i. + . . 341.33
i. + . . 298.67
i. + 2 . 256
s

t 0 100

```

```
i5 0 .333 .7 512 1
i. + . . 448
i. + . . 384
i. + . . 360
i. + . . 341.33
i. + . . 298.67
i. + 2 . 256
s
```

```
t 0 100
i6 0 .333 .7 512 1
i. + . . 448
i. + . . 384
i. + . . 360
i. + . . 341.33
i. + . . 298.67
i. + 2 . 256
```

e

```
</CsScore>
</CsoundSynthesizer>
```

## See Also

*chebyshevpoly, mac maca*

## Credits

Author: Anthony Kozar  
January 2008

New in Csound version 5.08

# port

`port` — Applies portamento to a step-valued control signal.

## Description

Applies portamento to a step-valued control signal.

## Syntax

```
kres port ksig, ihtim [, isig]
```

## Initialization

*ih*tim -- half-time of the function, in seconds.

*isig* (optional, default=0) -- initial (i.e. previous) value for internal feedback. The default value is 0. Negative value will cause initialization to be skipped and last value from previous instance to be used as initial value for note.

## Performance

*kres* -- the output signal at control-rate.

*ksig* -- the input signal at control-rate.

*port* applies portamento to a step-valued control signal. At each new step value, *ksig* is low-pass filtered to move towards that value at a rate determined by *ih*tim. *ih*tim is the “half-time” of the function (in seconds), during which the curve will traverse half the distance towards the new value, then half as much again, etc., theoretically never reaching its asymptote. With *portk*, the half-time can be varied at the control rate.

## Examples

Here is an example of the `port` opcode. It uses the file `port.csd` [examples/port.csd].

### Example 704. Example of the `port` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o port.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
0dbfs = 1
```

```
nchnls = 2

instr 1

aout diskin2 "fox.wav",1, 0, 1
kf,ka ptrack aout, 512 ; pitch track with winsize=1024
kcps port kf, 0.01 ; smooth freq
kamp port ka, 0.01 ; smooth amp
; drive an oscillator
asig poscil ampdb(kamp)*0dbfs, kcps, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
; simple sine wave
f 1 0 4096 10 1

i 1 0 5
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*areson, aresonk, atone, atonek, portk, reson, resonk, tone, tonek*

# portk

portk — Applies portamento to a step-valued control signal.

## Description

Applies portamento to a step-valued control signal.

## Syntax

```
kres portk ksig, khtim [, isig]
```

## Initialization

*isig* (optional, default=0) -- initial (i.e. previous) value for internal feedback. The default value is 0.

## Performance

*kres* -- the output signal at control-rate.

*ksig* -- the input signal at control-rate.

*khtim* -- half-time of the function in seconds.

*portk* is like *port* except the half-time can be varied at the control rate.

## Examples

Here is an example of the portk opcode. It uses the file *portk.csd* [examples/portk.csd].

### Example 705. Example of the portk opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          ; -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o portk.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 128
nchnls = 1

;Example by Andres Cabrera 2007

FLpanel "Slider", 650, 140, 50, 50
gkval1, gislider1 FLslider "Watch me", 0, 127, 0, 5, -1, 580, 30, 25, 20
gkval2, gislider2 FLslider "Move me", 0, 127, 0, 5, -1, 580, 30, 25, 80
```

```
gkhtim, gislider3 FLslider "khtim", 0.1, 1, 0, 6, -1, 30, 100, 610, 10
FLpanelEnd
FLrun

FLsetVal_i 0.1, gislider3 ;set initial time to 0.1

instr 1
kval portk gkval2, gkhtim ; take the value of slider 2 and apply portamento
FLsetVal 1, kval, gislider1 ;set the value of slider 1 to kval
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one minute.
i 1 0 60
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*areson, aresonk, atone, atonek, port, reson, resonk, tone, tonek*

## Credits

Author: Robin Whittle  
Australia  
May 1997

# poscil3

poscil3 — High precision oscillator with cubic interpolation.

## Description

High precision oscillator with cubic interpolation.

## Syntax

```
ares poscil3 aamp, acps [, ifn, iphs]
ares poscil3 aamp, kcps [, ifn, iphs]
ares poscil3 kamp, acps [, ifn, iphs]
ares poscil3 kamp, kcps [, ifn, iphs]
ires poscil3 kamp, kcps [, ifn, iphs]
kres poscil3 kamp, kcps [, ifn, iphs]
```

## Initialization

*ifn* -- (optional) function table number. This defaults to -1 which indicates a sinewave.

*iphs* (optional, default=0) -- initial phase (normalized table index 0-1). If a negative value is given initialisation of the phase is skipped.

## Performance

*ares* -- output signal

*kamp*, *aamp* -- the amplitude of the output signal.

*kcps*, *acps* -- the frequency of the output signal in cycles per second.

*poscil3* works like *poscil*, but uses cubic interpolation.

Note that *poscil3* can use deferred (non-power of two) length tables.

## Examples

Here is an example of the *poscil3* opcode. It uses the file *poscil3.csd* [examples/poscil3.csd].

### Example 706. Example of the *poscil3* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;;realtime audio out
```

```

;-iadc    ;;uncomment -iadc if real audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o poscil3.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine ftgen 0, 0, 2^10, 10, 1

instr 1

krnd randomh 40, 440, 1 ; produce random values
ain poscil3 .6, krnd, giSine
kline line 1, p3, 0 ; straight line
aL,aR pan2 ain, kline ; sent across image
outs aL, aR

endin
</CsInstruments>
<CsScore>
i1 0 10
e
</CsScore>
</CsoundSynthesizer>

```

Here is another example of the poscil3 opcode, which uses a table filled from a sound file. It uses the file *poscil3-file.csd* [examples/poscil3-file.csd].

### Example 707. Another example of the poscil3 opcode.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in   No messages
-odac          -iadc      -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o poscil3-file.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Example written by Joachim Heintz 07/2008

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; non-normalized function table with a sample 1
giFile ftgen 1, 0, 0, -1, "fox.wav", 0, 0, 0

; Instrument #1 - uses poscil3 for playing samples from a function table
instr 1
kamp = p4
kspeed = p5
ifn = 1
iskip = p6
kcps = kspeed / ((ftlen(ifn) / ftsr(ifn))); frequency of the oscillator
iphs = iskip / ((ftlen(ifn) / ftsr(ifn))); calculates skiptime to phase values (0-1)

a1 poscil3 kamp, kcps, ifn, iphs
out a1

```

```
endin
</CsInstruments>
<CsScore>
i1 0 2.756 1 1 0
i1 3 2.756 1 -1 0
i1 6 1.378 1 .5 2.067
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*poscil*

## Credits

Authors: John ffitich, Gabriel Maldonado  
Italy

New in Csound version 3.52

variants with a-rate amplitude or frequency new in 5.16

Function table optional in Csound version 6.00

Skip of initialisation new in version 6.06

# poscil

poscil — High precision oscillator.

## Description

High precision oscillator.

## Syntax

```
ares poscil aamp, acps [, ifn, iphs]
ares poscil aamp, kcps [, ifn, iphs]
ares poscil kamp, acps [, ifn, iphs]
ares poscil kamp, kcps [, ifn, iphs]
ires poscil kamp, kcps [, ifn, iphs]
kres poscil kamp, kcps [, ifn, iphs]
```

## Initialization

*ifn* -- (optional) function table number. This defaults to -1 which indicates a sinewave.

*iphs* (optional, default=0) -- initial phase (normalized table index 0-1). If a negative value is given initialisation of the phase is skipped.

## Performance

*ares* -- output signal

*kamp*, *aamp* -- the amplitude of the output signal.

*kcps*, *acps* -- the frequency of the output signal in cycles per second.

*poscil* (precise oscillator) is the same as *oscili*, but allows much more precise frequency control, especially when using long tables and low frequency values. It uses floating-point table indexing, instead of integer math, like *oscil* and *oscili*. It is only a bit slower than *oscili*.

Since Csound 4.22, *poscil* can accept also negative frequency values and use a-rate values both for amplitude and frequency. So both AM and FM are allowed using this opcode.

The opcode *poscil3* is the same as *poscil*, but uses cubic interpolation.

Note that *poscil* can use deferred (non-power of two) length tables.

## Examples

Here is an example of the *poscil* opcode. It uses the file *poscil.csd* [examples/poscil.csd].

### Example 708. Example of the *poscil* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o poscil.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

seed 0
gisine ftgen 0, 0, 2^10, 10, 1

instr 1

ipeak random 0, 1 ;where is the envelope peak
asig poscil .8, 220, gisine
aenv transeg 0, p3*ipeak, 6, 1, p3-p3*ipeak, -6, 0
aL,aR pan2 asig*aenv, ipeak ;pan according to random value
outs aL, aR

endin

</CsInstruments>
<CsScore>
i1 0 5
i1 4 5
i1 8 5
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*poscil3*

## Credits

Author: Gabriel Maldonado  
Italy  
1998

November 2002. Added a note about the changes to Csound version 4.22, thanks to Rasmus Ekman.

New in Csound version 3.52

Function table optional in Csound version 6.00

Skip of initialisation new in version 6.06

## pow

`pow` — Computes one argument to the power of another argument.

### Description

Computes *xarg* to the power of *kpow* (or *ipow*) and scales the result by *inorm*.

### Syntax

```
ares pow aarg, kpow [, inorm]
ires pow iarg, ipow [, inorm]
kres pow karg, kpow [,
      inorm]

ires[] pow iarg[], ipow[]
kres[] pow karg[], kpow[]

ires[] pow iarg[], ipow
kres[] pow karg[], kpow
```

### Initialization

*inorm* (optional, default=1) -- The number to divide the result (default to 1). This is especially useful if you are doing powers of a- or k- signals where samples out of range are extremely common!

### Performance

*aarg*, *iarg*, *karg* -- the base.

*ipow*, *kpow* -- the exponent.



#### Note

Use `^` with caution in arithmetical statements, as the precedence may not be correct. New in Csound version 3.493.

### Examples

Here is an example of the `pow` opcode. It uses the file `pow.csd` [examples/pow.csd].

#### Example 709. Example of the `pow` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
```

```
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
;-o pow.wav      ; output to audio file
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; Lo-Fi sound

kpow = 10      ;exponent
kbase line 1, p3, 1.4 ;vary the base
kQuantize pow kbase, kpow
kQuantize = kQuantize*0.5 ;half the number of steps for each side of a bipolar signal
printk2 kQuantize
asig disk2 "fox.wav", 1, 0, 1 ;loop the fox
asig = round(asig * kQuantize) / kQuantize ;quantize and scale audio signal
outs asig, asig

endin
</CsInstruments>
<CsScore>

i1 0 19.2

e
</CsScore>
</CsSoundSynthesizer>
```

Its output should include lines like these:

```
i1      0.50000
i1      0.50007
i1      0.50014
.....
i1     14.45986
i1     14.46130
```

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

# powershape

powershape — Waveshapes a signal by raising it to a variable exponent.

## Description

The *powershape* opcode raises an input signal to a power with pre- and post-scaling of the signal so that the output will be in a predictable range. It also processes negative inputs in a symmetrical way to positive inputs, calculating a dynamic transfer function that is useful for waveshaping.

## Syntax

```
aout powershape ain, kShapeAmount [, ifullscale]
```

## Initialization

*ifullscale* -- optional parameter specifying the range of input values from *-ifullscale* to *ifullscale*. Defaults to 1.0 -- you should set this parameter to the maximum expected input value.

## Performance

*ain* -- the input signal to be shaped.

*aout* -- the output signal.

*kShapeAmount* -- the amount of the shaping effect applied to the input; equal to the power that the input signal is raised.

The *powershape* opcode is very similar to the *pow* unit generators for calculating the mathematical "power of" operation. However, it introduces a couple of twists that can make it much more useful for waveshaping audio-rate signals. The *kShapeAmount* parameter is the exponent to which the input signal is raised.

To avoid discontinuities, the *powershape* opcode treats all input values as positive (by taking their absolute value) but preserves their original sign in the output signal. This allows for smooth shaping of any input signal while varying the exponent over any range. (*powershape* also (hopefully) deals intelligently with discontinuities that could arise when the exponent and input are both zero. Note though that negative exponents will usually cause the signal to exceed the maximum amplitude specified by the *ifullscale* parameter and should normally be avoided).

The other adaptation involves the *ifullscale* parameter. The input signal is divided by *ifullscale* before being raised to *kShapeAmount* and then multiplied by *ifullscale* before being output. This normalizes the input signal to the interval [-1,1], guaranteeing that the output (before final scaling) will also be within this range for positive shaping amounts and providing a smoothly evolving transfer function while varying the amount of shaping. Values of *kShapeAmount* between (0,1) will make the signal more "convex" while values greater than 1 will make it more "concave". A value of exactly 1.0 will produce no change in the input signal.

## See Also

*pow*, *powoftwo*

## Examples

Here is an example of the powershape opcode. It uses the file *powershape.csd* [examples/powershape.csd].

### Example 710. Example of the powershape opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o abs.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
instr 1
  imaxamp      =          10000
  kshapeamt    line      p5, p3, p6
  aosc         oscili     1.0, cpspch(p4), 1
  aout         powershape aosc, kshapeamt
  adeclick     linseg     0.0, 0.01, 1.0, p3 - 0.06, 1.0, 0.05, 0.0

          out          aout * adeclick * imaxamp
endin

</CsInstruments>
<CsScore>
f1 0 32768 10 1

i1 0 1      7.00  0.000001 0.8
i1 + 0.5    7.02  0.01    1.0
i1 + .      7.05  0.5     1.0
i1 + .      7.07  4.0     1.0
i1 + .      7.09  1.0    10.0
i1 + 2      7.06  1.0    25.0

e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Anthony Kozar  
January 2008

New in Csound version 5.08

# powoftwo

powoftwo — Performs a power-of-two calculation.

## Description

Performs a power-of-two calculation.

## Syntax

```
powoftwo(x) (init-rate or control-rate args only)
```

## Performance

*powoftwo()* function returns  $2^x$  and allows positive and negatives numbers as argument. The range of values admitted in *powoftwo()* is -5 to +5 allowing a precision more fine than one cent in a range of ten octaves. If a greater range of values is required, use the slower opcode *pow*.

These functions are fast, because they read values stored in tables. Also they are very useful when working with tuning ratios. They work at i- and k-rate.

## Examples

Here is an example of the powoftwo opcode. It uses the file *powoftwo.csd* [examples/powoftwo.csd].

### Example 711. Example of the powoftwo opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o poweroftwo.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; bit reduction for Lo-Fi sound

iBits = p5          ;bit depth
iQuantize = powoftwo(iBits) ;find number of discrete steps for this bit depth
iQuantize = iQuantize*0.5 ;half the number of steps for each side of a bipolar signal
print iQuantize
asig soundin "fox.wav"
asig = round(asig * iQuantize) / iQuantize ;quantize audio signal (bit reduce)
outs asig, asig
```

```
endin
</CsInstruments>
<CsScore>
;          bits
i1 0 3 16
i1 ^+3 . 12
i1 ^+3 . 8
i1 ^+3 . 4
i1 ^+3 . 2
i1 ^+3 . 1

e
</CsScore>
</CsoundSynthesizer>
```

Its output should include a line like this:

```
instr 1: iQuantize = 32768.000
instr 1: iQuantize = 2048.000
instr 1: iQuantize = 128.000
instr 1: iQuantize = 8.000
instr 1: iQuantize = 2.000
instr 1: iQuantize = 1.000
```

## See Also

*logbtwo, pow*

## Credits

Author: Gabriel Maldonado  
Italy  
June 1998

Author: John ffitch  
University of Bath, Codemist, Ltd.  
Bath, UK  
July 1999

New in Csound version 3.57

# prealloc

prealloc — Creates space for instruments but does not run them.

## Description

Creates space for instruments but does not run them.

## Syntax

```
prealloc insnum, icount
prealloc "insname", icount
```

## Initialization

*insnum* -- instrument number

*icount* -- number of instrument allocations

*"insname"* -- A string (in double-quotes) representing a named instrument.

## Performance

All instances of *prealloc* must be defined in the header section, not in the instrument body.

## Examples

Here is an example of the *prealloc* opcode. It uses the file *prealloc.csd* [examples/prealloc.csd].

### Example 712. Example of the prealloc opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o prealloc.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Pre-allocate memory for five instances of Instrument #1.
prealloc 1, 5
```

```
; Instrument #1
instr 1
  ; Generate a waveform, get the cycles per second from the 4th p-field.
  a1 oscil 6500, p4, 1
  out a1
endin

</CsInstruments>
<CsScore>

; Just generate a nice, ordinary sine wave.
f 1 0 32768 10 1

; Play five instances of Instrument #1 for one second.
; Note that 4th p-field contains cycles per second.
i 1 0 1 220
i 1 0 1 440
i 1 0 1 880
i 1 0 1 1320
i 1 0 1 1760
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*cpuprc, maxalloc*

## Credits

Author: Gabriel Maldonado  
Italy  
July 1999

Example written by Kevin Conder.

New in Csound version 3.57

# prepiano

prepiano — Creates a tone similar to a piano string prepared in a Cageian fashion.

## Description

Audio output is a tone similar to a piano string, prepared with a number of rubbers and rattles. The method uses a physical model developed from solving the partial differential equation.

## Syntax

```
ares prepiano ifreq, iNS, iD, iK, \  
    iT30, iB, kbcl, kbcr, imass, ihvfreq, iinit, ipos, ivel, isfreq, \  
    isspread[, irattles, irubbers]  
  
al,ar prepiano ifreq, iNS, iD, iK, \  
    iT30, iB, kbcl, kbcr, imass, ihvfreq, iinit, ipos, ivel, isfreq, \  
    isspread[, irattles, irubbers]
```

## Initialization

*ifreq* -- The base frequency of the string.

*iNS* -- the number of strings involved. In a real piano 1, 2 or 3 strings are found in different frequency regions.

*iD* -- the amount each string other than the first is detuned from the main frequency, measured in cents.

*iK* -- dimensionless stiffness parameter.

*iT30* -- 30 db decay time in seconds.

*ib* -- high-frequency loss parameter (keep this small).

*imass* -- the mass of the piano hammer.

*ihvfreq* -- the frequency of the natural vibrations of the hammer.

*iinit* -- the initial position of the hammer.

*ipos* -- position along the string that the strike occurs.

*ivel* -- normalized strike velocity.

*isfreq* -- scanning frequency of the reading place.

*isspread* -- scanning frequency spread.

*irattles* -- table number giving locations of any rattle(s).

*irubbers* -- table number giving locations of any rubbers(s).

The rattles and rubbers tables are collections of four values, preceded by a count. In the case of a rattle the four are position, mass density ratio of rattle/string, frequency of rattle and vertical length of the rattle.

For the rubber the fours are position, mass density ratio of rubber/string, frequency of rubber and the loss parameter.

## Performance

A note is played on a piano string, with the arguments as below.

*bcL* -- Boundary condition at left end of string (1 is clamped, 2 pivoting and 3 free).

*bcR* -- Boundary condition at right end of string (1 is clamped, 2 pivoting and 3 free).

Note that changing the boundary conditions during playing may lead to glitches and is made available as an experiment.

## Examples

Here is an example of the *prepiano* opcode. It uses the file *prepiano.csd* [examples/prepiano.csd].

### Example 713. Example of the *prepiano* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if real audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o prepiano.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
;;      fund NS detune stiffness decay loss (bndry) (hammer) scan prep
aa,ab prepiano 60, 3, 10, p4, 3, 0.002, 2, 2, 1, 5000, -0.01, p5, p6, 0, 0.1, 1, 2
outs aa*.2, ab*.2

endin
</CsInstruments>
<CsScore>
f1 0 8 2 1 0.6 10 100 0.001 ;; 1 rattle
f2 0 8 2 1 0.7 50 500 1000 ;; 1 rubber
i1 0.0 1 1 0.09 20
i1 1 . -1 0.09 40           ;; 1 -> skip initialisation
i1 2 . -1 0.09 60
i1 3 . -1 0.09 80
i1 4 1.8 -1 0.09 100
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Stefan Bilbao

University of Edinburgh, UK  
Author: John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 5.05

# print

`print` — Displays the values init (i-rate) variables.

## Description

These units will print orchestra init-values.

## Syntax

```
print iarg [, iarg1] [, iarg2] [...]
```

## Initialization

*iarg*, *iarg2*, ... -- i-rate arguments.

## Performance

`print` -- print the current value of the i-time arguments (or expressions) *iarg* at every i-pass through the instrument.



### Note

The `print` opcode will truncate decimal places and may not show the complete value. Csound's precision depends on whether it is the floats (32-bit) or double (64-bit) *version*, since most internal calculations use one of these formats. If you need more resolution in the console output, you can try `printf`.

## Examples

Here is an example of the `print` opcode. It uses the file `print.csd` [examples/print.csd].

### Example 714. Example of the `print` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o print.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
```

```
irand = rnd(3) ;generate a random number from 0 to 3
print irand ;print it
asig poscil .7, 440*irand, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
f1 0 16384 10 1 ;sine wave

i 1 0 1
i 1 2 1
i 1 4 1
e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like these:

```
instr 1: irand = 2.930
instr 1: irand = 0.420
instr 1: irand = 2.316
```

## See Also

*dispfft, display, printk, printk2, printks, printf and prints*

# printf

printf — printf-style formatted output

## Description

**printf** and **printf\_i** write formatted output, similarly to the C function printf(). **printf\_i** runs at i-time only, while **printf** runs both at initialization and performance time.

## Syntax

```
printf_i Sfmt, itrig, [iarg1[, iarg2[, ... ]]]
```

```
printf Sfmt, ktrig, [xarg1[, xarg2[, ... ]]]
```

## Initialization

*Sfmt* -- format string, has the same format as in printf() and other similar C functions, except length modifiers (l, ll, h, etc.) are not supported. The following conversion specifiers are allowed:

- d, i, o, u, x, X, e, E, f, F, g, G, c, s

*iarg1*, *iarg2*, ... -- input arguments (max. 30) for format. Integer formats like %d round the input values to the nearest integer.

*itrig* -- if greater than zero the opcode performs the printing; otherwise it is a null operation.

## Performance

*ktrig* -- if greater than zero and different from the value on the previous control cycle the opcode performs the requested printing. Initially this previous value is taken as zero.

*xarg1*, *xarg2*, ... -- input arguments (max. 30) for format. Integer formats like %d round the input values to the nearest integer. Note that only k-rate and i-rate arguments are valid (no a-rate printing)

## Examples

Here is an example of the printf opcode. It uses the file *printf.csd* [examples/printf.csd].

### Example 715. Example of the printf opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
;-o printf.wav -W ;; for file output any platform
</CsOptions>
```

```
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
Odbfs = 1

instr 1
Sfile strget p4
ivld filevalid Sfile

if ivld=0 then
    printf_i "Audiofile '%s' does not exist!\n", 1, Sfile
else
    asig disk2 Sfile, 1
    outs asig, asig
endif

endin

</CsInstruments>
<CsScore>

i 1 0 3 "frox.wav";file does not exist!!!
i 1 + 3 "fox.wav";but this one certainly does...

e
</CsScore>
</CsoundSynthesizer>
```

The example will produce the following output:

```
Audiofile 'frox.wav' does not exist!
```

## See Also

More information about printf: <http://www.cplusplus.com/reference/clibrary/cstdio/printf/>

## Credits

Author: Istvan Varga  
2005

# printk2

printk2 — Prints a new value every time a control variable changes.

## Description

Prints a new value every time a control variable changes.

## Syntax

```
printk2 kvar [, inumspaces]
```

## Initialization

*inumspaces* (optional, default=0) -- number of space characters printed before the value of *kvar*

## Performance

*kvar* -- signal to be printed

Derived from Robin Whittle's *printk*, prints a new value of *kvar* each time *kvar* changes. Useful for monitoring MIDI control changes when using sliders.



### Warning

**WARNING!** Don't use this opcode with normal, continuously variant k-signals, because it can hang the computer, as the rate of printing is too fast.

## Examples

Here is an example of the printk2 opcode. It uses the file *printk2.csd* [examples/printk2.csd].

### Example 716. Example of the printk2 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o printk2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
```

```
kval  randomh 0, 1.2, 20 ;choose between 0 and 1.2
if kval >0 && kval<=.5 then ;3 possible outcomes
  kval = 1
elseif kval >.5 && kval<=1 then
  kval =2
elseif kval >1 then
  kval =3
endif

printk2 kval ;print value when it changes
asig  poscil .7, 440*kval, 1
      outs asig, asig

endin
</CsInstruments>
<CsScore>
f1 0 16384 10 1

i 1 0 5
e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like these:

```
i1      0.00000
i1      3.00000
i1      1.00000
i1      3.00000
i1      1.00000
i1      2.00000
i1      3.00000
.....
```

## See Also

*printk* and *printks*

## Credits

Author: Gabriel Maldonado  
Italy  
1998

New in Csound version 3.48

# printk

printk — Prints one k-rate value at specified intervals.

## Description

Prints one k-rate value at specified intervals.

## Syntax

```
printk itime, kval [, ispace]
```

## Initialization

*itime* -- time in seconds between printings.

*ispace* (optional, default=0) -- number of spaces to insert before printing. (default: 0, max: 130)

## Performance

*kval* -- The k-rate values to be printed.

*printk* prints one k-rate value on every k-cycle, every second or at intervals specified. First the instrument number is printed, then the absolute time in seconds, then a specified number of spaces, then the *kval* value. The variable number of spaces enables different values to be spaced out across the screen - so they are easier to view.

This opcode can be run on every k-cycle it is run in the instrument. To every accomplish this, set *itime* to 0.

When *itime* is not 0, the opcode print on the first k-cycle it is called, and subsequently when every *itime* period has elapsed. The time cycles start from the time the opcode is initialized - typically the initialization of the instrument.

## Examples

Here is an example of the printk opcode. It uses the file *printk.csd* [examples/printk.csd].

### Example 717. Example of the printk opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o printk.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```
; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1.
instr 1
  ; Change a value linearly from 0 to 100,
  ; over the period defined by p3.
  kval line 0, p3, 100

  ; Print the value of kval, once per second.
  printk 1, kval
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for 5 seconds.
i 1 0 5
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
i 1 time 0.00002: 0.00000
i 1 time 1.00002: 20.01084
i 1 time 2.00002: 40.02999
i 1 time 3.00002: 60.04914
i 1 time 4.00002: 79.93327
```

## See Also

*printk2* and *printks*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

Thanks goes to Luis Jure for pointing out a mistake with the *itime* parameter.

# printks

printks — Prints at k-rate using a printf() style syntax.

## Description

Prints at k-rate using a printf() style syntax.

## Syntax

```
printks "string", itime [, kval1] [, kval2] [...]
```

## Initialization

*"string"* -- the text string to be printed. Can be up to 8192 characters and must be in double quotes.

*itime* -- time in seconds between printings.

## Performance

*kval1*, *kval2*, ... (optional) -- The k-rate values to be printed. These are specified in *"string"* with the standard C value specifier (%f, %d, etc.) in the order given.

In Csound version 4.23, you can use as many *kval* variables as you like. In versions prior to 4.23, you must specify 4 and only 4 *kvals* (using 0 for unused *kvals*).

*printks* prints numbers and text which can be i-time or k-rate values. *printks* is highly flexible, and if used together with cursor positioning codes, could be used to write specific values to locations in the screen as the Csound processing proceeds.

A special mode of operation allows this *printks* to convert *kval1* input parameter into a 0 to 255 value and to use it as the first character to be printed. This enables a Csound program to send arbitrary characters to the console. To achieve this, make the first character of the string a # and then, if desired continue with normal text and format specifiers.

This opcode can be run on every k-cycle it is run in the instrument. To every accomplish this, set *itime* to 0.

When *itime* is not 0, the opcode print on the first k-cycle it is called, and subsequently when every *itime* period has elapsed. The time cycles start from the time the opcode is initialized - typically the initialization of the instrument.

## Print Output Formatting

All standard C language printf() control characters may be used. For example, if *kval1* = 153.26789 then some common formatting options are:

1. %f prints with full precision: 153.26789
2. %5.2f prints: 153.26
3. %d prints integers-only: 153

4. %c treats *kval1* as an ascii character code.

In addition to all the printf() codes, printks supports these useful character codes:

printks Code	Character Code
\\r, \\R, %r, or %R	return character (\r)
\\n, \\N, %n, %N	newline character (\n)
\\t, \\T, %t, or %T	tab character (\t)
%;	semicolon character (;) This was needed because a “;” is interpreted as an comment.
^	escape character (0x1B)
^ ^	caret character (^)
~	ESC[ (escape+[ is the escape sequence for ANSI consoles)
~	tilde (~)

For more information about printf() formatting, consult any C language documentation.



## Note

Prior to version 4.23, only the %f format code was supported.

## Examples

Here is an example of the printks opcode. It uses the file *printks.csd* [examples/printks.csd].

### Example 718. Example of the printks opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o printks.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1.
instr 1
; Change a value linearly from 0 to 100,
; over the period defined by p3.
kup line 0, p3, 100
; Change a value linearly from 30 to 10,
; over the period defined by p3.
kdown line 30, p3, 10
```

```
    ; Print the value of kup and kdown, once per second.
    printks "kup = %f, kdown = %f\\n", 1, kup, kdown
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for 5 seconds.
i 1 0 5
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
kup = 0.000000, kdown = 30.000000
kup = 20.010843, kdown = 25.962524
kup = 40.029991, kdown = 21.925049
kup = 60.049141, kdown = 17.887573
kup = 79.933266, kdown = 13.872493
```

## See Also

*printk2* and *printk*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

Thanks goes to Luis Jure for pointing out a mistake with the *itime* parameter.

Thanks to Matt Ingalls, updated the documentation for version 4.23.

# printks2

printks2 — Prints a new value every time a control variable changes using a printf() style syntax.

## Description

Prints a new value every time a control variable changes using a printf() style syntax.

## Syntax

```
printks2 "string", kval
```

## Initialization

"string" -- the text string to be used as a format.

## Performance

kval -- signal to be printed. The style of printing is specified in "string" with the standard C value specifier (%f, %d, etc.).

## Print Output Formatting

All standard C language printf() control characters may be used. For example, if *kval* = 153.26789 then some common formatting options are:

1. %f prints with full precision: 153.26789
2. %5.2f prints: 153.26
3. %d prints integers-only: 153
4. %c treats *kval* as an ascii character code.

In addition to all the printf() codes, printks2 supports these useful character codes:

printks2 Code	Character Code
\\r, \\R, %r, or %R	return character (\r)
\\n, \\N, %n, %N	newline character (\n)
\\t, \\T, %t, or %T	tab character (\t)
%;	semicolon character (;) This was needed because a ";" is interpreted as an comment.
^	escape character (0x1B)
^ ^	caret character (^)
~	ESC[ (escape+[ is the escape sequence for ANSI consoles)
~~	tilde (~)

For more information about printf() formatting, consult any C language documentation.

## Examples

Here is an example of the printks2 opcode. It uses the file *printks2.csd* [examples/printks2.csd].

### Example 719. Example of the printks2 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
; For Non-realtime ouput leave only the line below:
; -o printk.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1.
instr 1
; Change a value linearly from 0 to 100,
; over the period defined by p3.
kval line 0, p3, 100

; Print the value of kval when it changes.
printks2 "value now %f\n", int(kval)
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for 5 seconds.
i 1 0 5
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*printk2* and *printk2* and *printk*

## Credits

Author: John ffitth  
Bath, UK  
Mar 2014

New in Csound version 6.03

# prints

prints — Prints at init-time using a printf() style syntax.

## Description

Prints at init-time using a printf() style syntax.

## Syntax

```
prints "string" [, kval1] [, kval2] [...]
```

## Initialization

"string" -- the text string to be printed. Can be up to 8192 characters and must be in double quotes.

## Performance

*kval1*, *kval2*, ... (optional) -- The k-rate values to be printed. These are specified in "string" with the standard C value specifier (%f, %d, etc.) in the order given. Use 0 for those which are not used.

*prints* is similar to the *printks* opcode except it operates at init-time instead of k-rate. For more information about output formatting, please look at *printks's documentation*.

## Examples

Here is an example of the prints opcode. It uses the file *prints.csd* [examples/prints.csd].

### Example 720. Example of the prints opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o prints.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

/* Written by Matt Ingalls, edited by Kevin Conder. */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Init-time print.
prints "%2.3f\\t%!%!%!%!%;semicolon! %%\\n", 1234.56789
```

```
endin

</CsInstruments>
<CsScore>

/* Written by Matt Ingalls, edited by Kevin Conder. */
; Play instrument #1.
i 1 0 0.004

</CsScore>
</CsoundSynthesizer>
```

Its output should include a line like this:

```
1234.568      ;;;;semicolons!
```

## See Also

*prints*

## Credits

Author: Matt Ingalls  
January 2003

# product

product — Multiplies any number of a-rate signals.

## Description

Multiplies any number of a-rate signals.

## Syntax

```
ares product asig1, asig2 [, asig3] [...]
```

## Performance

*asig1, asig2, asig3, ...* -- a-rate signals to be multiplied.

## Examples

Here is an example of the product opcode. It uses the file *product.csd* [examples/product.csd].

### Example 721. Example of the product opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsoundOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o product.wav -W ;; for file output any platform
</CsoundOptions>
<CsoundInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gisine ftgen 0, 0, 2^10, 10, 1

instr 1

a1  oscili 1, 10.0, gisine ;combine 3 sinusses
a2  oscili 1, 1.0, gisine ;at different rates
a3  oscili 1, 3.0, gisine
ares product a1, a2, a3

ares = ares*10000 ;scale result and
asig poscil .5, ares+110, gisine ;add to frequency
outs asig, asig

endin
</CsoundInstruments>
<CsoundScore>
```

```
i1 0 5  
e  
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Author: Gabriel Maldonado  
Italy  
April 1999

New in Csound version 3.54

# product

product — Calculates the product of an array.

## Description

Takes a numeric array (k or i-rate) and calculates its product.

## Syntax

```
kres/iresproduct karr[]/iarr[] (k- or i-arrays )
```

## Examples

Here is an example of the product opcode. It uses the file *producta.csd* [examples/productarray.csd].

### Example 722. Example of the product opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsoundOptions>
-n
</CsoundOptions>
<CsoundInstruments>

instr 1
  kArr1[] fillarray 1,3,2,7,4
  k1 product kArr1
  printk2 k1
  turnoff
endin

</CsoundInstruments>
<CsoundScore>
i1 0 1
</CsoundScore>
</CsoundSynthesizer>
```

## See Also

*dot sumarray*

## Credits

Author: Victor Lazzarini

New in version 6.09

# pset

pset — Defines and initializes numeric arrays at orchestra load time.

## Description

Defines and initializes numeric arrays at orchestra load time.

## Syntax

```
pset icon1 [, icon2] [...]
```

## Initialization

*icon1*, *icon2*, ... -- preset values for a MIDI instrument

*pset* (optional) defines and initializes numeric arrays at orchestra load time. It may be used as an orchestra header statement (i.e. instrument 0) or within an instrument. When defined within an instrument, it is not part of its i-time or performance operation, and only one statement is allowed per instrument. These values are available as i-time defaults. When an instrument is triggered from MIDI it only gets p1 and p2 from the event, and p3, p4, etc. will receive the actual preset values.

## Examples

Here is an example of the pset opcode. It uses the file *pset.csd* [examples/pset.csd]

### Example 723. Example of the pset opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pset.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 2

instr 1 ;this shows an example with non-midi use

pset 1, 0, 1, 220, 0.5
asig poscil p5, p4, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
```

```

f 1 0 1024 10 1 ;sine wave

i 1 0 1
i 1 1 1 440
i 1 2 1 440 0.1
e
</CsScore>
</CsoundSynthesizer>

```

Here is another example of the pset opcode, using pset with midi. It uses the file *pset-midi.csd* [examples/pset-midi.csd]

### Example 724. Second example of the pset opcode.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0 --midi-key-oct=4 --midi-velocity=5   ;;realtime audio out and virtual midi
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pset-midi.wav -W   ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 2

instr 1

      pset 0, 0, 3600, 0, 0, 0
instrument = p1
istarttime = p2
iattack    = 0.005
isustain   = p3
irelease   = 0.06
p3         = isustain + iattack + irelease
kdamping   = linsegr 0.0, iattack, 1.0, isustain, 1.0, irelease, 0.0

ioctave    = p4
ifrequency = cpsoct(ioctave)
iamplitude = p5*.15   ;lower volume

print p1, p2, p3, p4, p5
asig STKBandedWG ifrequency, iamplitude
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 0 60 ; runs 69 seconds
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*strset*

# ptable

ptable — Accesses table values by direct indexing.

## Description

Accesses table values by direct indexing.

## Syntax

```
ares ptable andx, ifn [, ixmode] [, ixoff] [, iwrap]
ires ptable indx, ifn [, ixmode] [, ixoff] [, iwrap]
kres ptable kndx, ifn [, ixmode] [, ixoff] [, iwrap]
```

## Initialization

*ifn* -- function table number.

*ixmode* (optional) -- index data mode. The default value is 0.

- 0 = raw index
- 1 = normalized (0 to 1)

*ixoff* (optional) -- amount by which index is to be offset. For a table with origin at center, use `tablesize/2` (raw) or `.5` (normalized). The default value is 0.

*iwrap* (optional) -- wraparound index flag. The default value is 0.

- 0 = nowrap (index < 0 treated as index=0; index > tablesize sticks at index=size)
- 1 = wraparound.

## Performance

*ptable* invokes table lookup on behalf of init, control or audio indices. These indices can be raw entry numbers (0, 1, 2,... size - 1) or scaled values (0 to 1). Indices are first modified by the offset value then checked for range before table lookup (see *iwrap*). If index is likely to be full scale, or if interpolation is being used, the table should have an extended guard point. *table* indexed by a periodic phasor ( see *phasor*) will simulate an oscillator.

## Examples

Here is an example of the *ptable* opcode. It uses the file *ptable.csd* [examples/ptable.csd].

### Example 725. Example of the *ptable* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o table.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Vary our index linearly from 0 to 1.
kndx line 0, p3, 1

; Read Table #1 with our index.
ifn = 1
ixmode = 1
kfreq ptable kndx, ifn, ixmode

; Generate a sine waveform, use our table values
; to vary its frequency.
a1 oscil 20000, kfreq, 2
out a1
endin

</CsInstruments>
<CsScore>

; Table #1, a line from 200 to 2,000.
f 1 0 1025 -7 200 1024 2000
; Table #2, a sine wave.
f 2 0 16384 10 1

; Play Instrument #1 for 2 seconds.
i 1 0 2
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*table, tablei, table3, ptable3, ptablei, oscil1, oscilli, osciln*

## Credits

Author: John ffitth  
Jan 2012

New in Csound version 5.16

# ptablei

ptablei — Accesses table values by direct indexing with linear interpolation.

## Description

Accesses table values by direct indexing with linear interpolation.

## Syntax

```
ares ptablei andx, ifn [, ixmode] [, ixoff] [, iwrap]
ires ptablei indx, ifn [, ixmode] [, ixoff] [, iwrap]
kres ptablei kndx, ifn [, ixmode] [, ixoff] [, iwrap]
```

## Initialization

*ifn* -- function table number. The table need not be a power of 2 long.

*ixmode* (optional) -- index data mode. The default value is 0.

- 0 = raw index
- 1 = normalized (0 to 1)

*ixoff* (optional) -- amount by which index is to be offset. For a table with origin at center, use `tablesize/2` (raw) or `.5` (normalized). The default value is 0.

*iwrap* (optional) -- wraparound index flag. The default value is 0.

- 0 = nowrap (index < 0 treated as index=0; index > tablesize sticks at index=size)
- 1 = wraparound.

## Performance

*ptablei* is a interpolating unit in which the fractional part of index is used to interpolate between adjacent table entries. The smoothness gained by interpolation is at some small cost in execution time (see also *oscili*, etc.), but the interpolating and non-interpolating units are otherwise interchangeable.

## Examples

Here is an example of the *ptablei* opcode. It uses the file *ptablei.csd* [examples/ptablei.csd].

### Example 726. Example of the *ptablei* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;;realtime audio out
```

```

;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o tablei.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

seed 0 ;generate new values every time the instr is played

instr 1

ifn = p4
isize = p5
ithresh = 0.5

itemp ftgen ifn, 0, isize, 21, 2

iwrite_value = 0
i_index = 0

loop_start:
  iread_value ptablei i_index, ifn

  if iread_value > ithresh then
    iwrite_value = 1
  else
    iwrite_value = -1
  endif
ptableiw iwrite_value, i_index, ifn
loop_lt i_index, 1, isize, loop_start
turnoff

endin

instr 2

ifn = p4
isize = ftlen(ifn)
prints "Index\tValue\n"

i_index = 0
loop_start:
  ivalue tablei i_index, ifn
  prints "%d:\t%f\n", i_index, ivalue

  loop_lt i_index, 1, isize, loop_start ;read table 1 with our index

aout oscili .5, 100, ifn ;use table to play the polypulse
outs aout, aout

endin
</CsInstruments>
<CsScore>
i 1 0 1 100 16
i 2 0 2 100
e
</CsScore>
</CsSoundSynthesizer>

```

## See Also

*table, tablei, table3, ptable, ptable3, oscil1, oscilli, osciln*

More information on this opcode: <http://www.csoundjournal.com/issue12/genInstruments.html> , written by Jacob Joaquin

## Credits

Author: John ffitch  
Jan 2012

New in Csound version 5.16

# ptable3

`ptable3` — Accesses table values by direct indexing with cubic interpolation.

## Description

Accesses table values by direct indexing with cubic interpolation.

## Syntax

```
ares ptable3 andx, ifn [, ixmode] [, ixoff] [, iwrap]
ires ptable3 indx, ifn [, ixmode] [, ixoff] [, iwrap]
kres ptable3 kndx, ifn [, ixmode] [, ixoff] [, iwrap]
```

## Initialization

*ifn* -- function table number.

*ixmode* (optional) -- index data mode. The default value is 0.

- 0 = raw index
- 1 = normalized (0 to 1)

*ixoff* (optional) -- amount by which index is to be offset. For a table with origin at center, use `tablesize/2` (raw) or `.5` (normalized). The default value is 0.

*iwrap* (optional) -- wraparound index flag. The default value is 0.

- 0 = nowrap (index < 0 treated as index=0; index > tablesize sticks at index=size)
- 1 = wraparound.

## Performance

`ptable3` is identical to `table3`, except that it uses does not require the table to have a power of two size.

## See Also

`table`, `tablei`, `table3`, `ptable`, `ptablei`, `oscill`, `oscilli`, `osciln`

## Credits

Author: John ffitch  
Jan 2012

New in Csound version 5.16

# ptablew

ptablew — Change the contents of existing function tables of any length.

## Description

This opcode operates on existing function tables, changing their contents. *ptablew* is for writing at k- or at a-rates, with the table number being specified at init time. Using *ptablew* with i-rate signal and index values is allowed, but the specified data will always be written to the function table at k-rate, not during the initialization pass. The valid combinations of variable types are shown by the first letter of the variable names.

## Syntax

```
ptablew asig, andx, ifn [, ixmode] [, ixoff] [, iwgmode]
```

```
ptablew isig, indx, ifn [, ixmode] [, ixoff] [, iwgmode]
```

```
ptablew ksig, kndx, ifn [, ixmode] [, ixoff] [, iwgmode]
```

## Initialization

*asig, isig, ksig* -- The value to be written into the table.

*andx, indx, kndx* -- Index into table, either a positive number range matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* != 0)

*ifn* -- Table number. Must be  $\geq 1$ . Floats are rounded down to an integer. If a table number does not point to a valid table, or the table has not yet been loaded (*GENOI*) then an error will result and the instrument will be de-activated.

*ixmode* (optional, default=0) -- index mode.

- 0 = *xndx* and *ixoff* ranges match the length of the table.
- !=0 = *xndx* and *ixoff* have a 0 to 1 range.

*ixoff* (optional, default=0) -- index offset.

- 0 = Total index is controlled directly by *xndx*, i.e. the indexing starts from the start of the table.
- !=0 = Start indexing from somewhere else in the table. Value must be positive and less than the table length (*ixmode* = 0) or less than 1 (*ixmode* != 0).

*iwgmode* (optional, default=0) -- Wrap and guardpoint mode.

- 0 = Limit mode.
- 1 = Wrap mode.
- 2 = Guardpoint mode.

## Performance

### Limit mode (0)

Limit the total index ( $ndx + ixoff$ ) to between 0 and the guard point. For a table of length 5, this means that locations 0 to 3 and location 4 (the guard point) can be written. A negative total index writes to location 0.

### Wrap mode (1)

Wrap total index value into locations 0 to E, where E is one less than the table length. For example, wrap into a 0 to 3 range - so that total index 6 writes to location 2.

### Guardpoint mode (2)

The guardpoint is written at the same time as location 0 is written - with the same value.

This facilitates writing to tables which are intended to be read with interpolation for producing smooth cyclic waveforms. In addition, before it is used, the total index is incremented by half the range between one location and the next, before being rounded down to the integer address of a table location.

Normally ( $igwmode = 0$  or  $1$ ) for a table of length 5 - which has locations 0 to 3 as the main table and location 4 as the guard point, a total index in the range of 0 to 0.999 will write to location 0. ("0.999" means just less than 1.0.) 1.0 to 1.999 will write to location 1 etc. A similar pattern holds for all total indexes 0 to 4.999 ( $igwmode = 0$ ) or to 3.999 ( $igwmode = 1$ ).  $igwmode = 0$  enables locations 0 to 4 to be written - with the guardpoint (4) being written with a potentially different value from location 0.

With a table of length 5 and the  $iwgmode = 2$ , then when the total index is in the range 0 to 0.499, it will write to locations 0 and 4. Range 0.5 to 1.499 will write to location 1 etc. 3.5 to 4.0 will also write to locations 0 and 4.

This way, the writing operation most closely approximates the results of interpolated reading. Guard point mode should only be used with tables that have a guardpoint.

Guardpoint mode is accomplished by adding 0.5 to the total index, rounding to the next lowest integer, wrapping it modulo the factor of two which is one less than the table length, writing the table (locations 0 to 3 in our example) and then writing to the guard point if index = 0.

*ptablew* has no output value. The last three parameters are optional and have default values of 0.

## Caution with k-rate table numbers

At k-rate or a-rate, if a table number of  $< 1$  is given, or the table number points to a non-existent table, or to one which has a length of 0 (it is to be loaded from a file later) then an error will result and the instrument will be deactivated. *kfn* and *afn* must be initialized at the appropriate rate using *init*. Attempting to load an i-rate value into *kfn* or *afn* will result in an error.



### Warning

Note that *ptablew* is always a k-rate opcode. This means that even its i-rate version runs at k-rate and will write the value of the i-rate variable. For this reason, the following code will not work as expected:

```
<CsoundSynthesizer>  
<CsoundOptions>
```

```
</CsOptions>
<CsInstruments>
giFt ftgen 1, 0, 8, 2, 0
instr 1
indx = 0
    ptablew 10, indx, giFt
ival tab_i indx, giFt
    print ival
endin
</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsoundSynthesizer>
```

Although it may seem this program should print a 10 to the console. It will print 0, because *tab\_i* will read the value at the initialization of the note, before the first performance pass, when *ptablew* writes its value.

## See Also

*tableiw*, *tablewkt*

## Credits

Author: John ffitc after Robin Whittle  
Feb 2012

# ptrack

ptrack — Tracks the pitch of a signal.

## Description

*ptrack* takes an input signal, splits it into *ihopsize* blocks and using a STFT method, extracts an estimated pitch for its fundamental frequency as well as estimating the total amplitude of the signal in dB, relative to full-scale (0dB). The method implies an analysis window size of  $2*ihopsize$  samples (overlapping by 1/2 window), which has to be a power-of-two, between 128 and 8192 (hopsizes between 64 and 4096). Smaller windows will give better time precision, but worse frequency accuracy (esp. in low fundamentals). This opcode is based on an original algorithm by M. Puckette.

## Syntax

```
kcps, kamp ptrack asig, ihopsize[,ipeaks]
```

## Initialization

*ihopsize* -- size of the analysis 'hop', in samples, required to be power-of-two (min 64, max 4096). This is the period between measurements.

*ipeaks, ihi* -- number of spectral peaks to use in the analysis, defaults to 20 (optional)

## Performance

*kcps* -- estimated pitch in Hz.

*kamp* -- estimated amplitude in dB relative to full-scale (0dB) (ie. always  $\leq 0$ ).

*ptrack* analyzes the input signal, *asig*, to give a pitch/amplitude pair of outputs, for the fundamental of a monophonic signal. The output is updated every  $sr/ihopsize$  seconds.

## Examples

Here is an example of the ptrack opcode. This example uses the files *ptrack.csd* [examples/ptrack.csd].

### Example 727. Example of the ptrack opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o ptrack.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```
sr = 44100
ksmps = 32
Odbfs = 1
nchnls = 2

instr 1

ihop = p4
aout diskin2 "fox.wav",1, 0, 1
kf,ka ptrack aout, ihop ; pitch track with different hopsizes
kcps port kf, 0.01 ; smooth freq
kamp port ka, 0.01 ; smooth amp
; drive an oscillator
asig poscil ampdb(kamp)*Odbfs, kcps, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
; simple sine wave
f 1 0 4096 10 1

i 1 0 5 128
i 1 6 5 512
i 1 12 5 1024
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
NUI, Maynooth.  
Maynooth, Ireland  
March, 2007

New in Csound version 5.05

# puts

puts — Print a string constant or variable

## Description

puts prints a string with an optional newline at the end whenever the trigger signal is positive and changes.

## Syntax

```
puts Sstr, ktrig[, inonl]
```

## Initialization

*Sstr* -- string to be printed

*inonl* (optional, defaults to 0) -- if non-zero, disables the default printing of a newline character at the end of the string

## Performance

*ktrig* -- trigger signal, should be valid at i-time. The string is printed at initialization time if *ktrig* is positive, and at performance time whenever *ktrig* is both positive and different from the previous value. Use a constant value of 1 to print once at note initialization.

## Examples

Here is an example of the puts opcode. It uses the file *puts.csd* [examples/puts.csd].

### Example 728. Example of the puts opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o puts.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kcount init 440
ktrig metro 10
kcount = kcount + ktrig
```

```
String sprintfk "frequency in Hertz : %d \n", kcount
puts String, kcount
asig poscil .7, kcount, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
f1 0 16384 10 1

i 1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like these:

```
frequency in Hertz : 440
frequency in Hertz : 441
frequency in Hertz : 442
....
frequency in Hertz : 459
frequency in Hertz : 460
```

## Credits

Author: Istvan Varga  
2005

# pvadd

`pvadd` — Reads from a *pvoc* file and uses the data to perform additive synthesis.

## Description

*pvadd* reads from a *pvoc* file and uses the data to perform additive synthesis using an internal array of interpolating oscillators. The user supplies the wave table (usually one period of a sine wave), and can choose which analysis bins will be used in the re-synthesis.

## Syntax

```
ares pvadd ktempnt, kfmmod, ifilcod, ifn, ibins [, ibinoffset] \  
      [, ibinincr] [, iextractmode] [, ifreqlim] [, igatefn]
```

## Initialization

*ifilcod* -- integer or character-string denoting a control-file derived from *pvana* analysis of an audio signal. An integer denotes the suffix of a file *pvoc.m*; a character-string (in double quotes) gives a filename, optionally a full pathname. If not fullpath, the file is sought first in the current directory, then in the one given by the environment variable *SADIR* (if defined). *pvoc* control files contain data organized for fft resynthesis. Memory usage depends on the size of the files involved, which are read and held entirely in memory during computation but are shared by multiple calls (see also *lpread*).

*ifn* -- table number of a stored function containing a sine wave.

*ibins* -- number of bins that will be used in the resynthesis (each bin counts as one oscillator in the resynthesis)

*ibinoffset* (optional) -- is the first bin used (it is optional and defaults to 0).

*ibinincr* (optional) -- sets an increment by which *pvadd* counts up from *ibinoffset* for *ibins* components in the re-synthesis (see below for a further explanation).

*iextractmode* (optional) -- determines if spectral extraction will be carried out and if so whether components that have changes in frequency below *ifreqlim* or above *ifreqlim* will be discarded. A value for *iextractmode* of 1 will cause *pvadd* to synthesize only those components where the frequency difference between analysis frames is greater than *ifreqlim*. A value of 2 for *iextractmode* will cause *pvadd* to synthesize only those components where the frequency difference between frames is less than *ifreqlim*. The default values for *iextractmode* and *ifreqlim* are 0, in which case a simple resynthesis will be done. See examples below.

*igatefn* (optional) -- is the number of a stored function which will be applied to the amplitudes of the analysis bins before resynthesis takes place. If *igatefn* is greater than 0 the amplitudes of each bin will be scaled by *igatefn* through a simple mapping process. First, the amplitudes of all of the bins in all of the frames in the entire analysis file are compared to determine the maximum amplitude value. This value is then used create normalized amplitudes as indices into the stored function *igatefn*. The maximum amplitude will map to the last point in the function. An amplitude of 0 will map to the first point in the function. Values between 0 and 1 will map accordingly to points along the function table. This will be made clearer in the examples below.

## Performance

*ktempnt* and *kfmmod* are used in the same way as in *pvoc*.

## Examples

```
ktime line 0, p3, p3
asig pvadd ktime, 1, "oboe.pvoc", 1, 100, 2
```

In the above, *ibins* is 100 and *ibinoffset* is 2. Using these settings the resynthesis will contain 100 components beginning with bin #2 (bins are counted starting with 0). That is, resynthesis will be done using bins 2-101 inclusive. It is usually a good idea to begin with bin 1 or 2 since the 0th and often 1st bin have data that is neither necessary nor even helpful for creating good clean resynthesis.

```
ktime line 0, p3, p3
asig pvadd ktime, 1, "oboe.pvoc", 1, 100, 2, 2
```

The above is the same as the previous example with the addition of the value 2 used for the optional *ibinincr* argument. This result will still result in 100 components in the resynthesis, but *pvadd* will count through the bins by 2 instead of by 1. It will use bins 2, 4, 6, 8, 10, and so on. For *ibins*=100, *ibinoffset*=10, and *ibinincr*=10, *pvadd* would use bins 10, 20, 30, 40, up to and including 100.

Below is an example using spectral extraction. In this example *iextractmode* is 1 and *ifreqlim* is 9. This will cause *pvadd* to synthesize only those bins where the frequency deviation, averaged over 6 frames, is greater than 9.

```
ktime line 0, p3, p3
asig pvadd ktime, 1, "oboe.pvoc", 1, 100, 2, 2, 1, 9
```

If *iextractmode* were 2 in the above, then only those bins with an average frequency deviation of less than 9 would be synthesized. If tuned correctly, this technique can be used to separate the pitched parts of the spectrum from the noisy parts. In practice this depends greatly on the type of sound, the quality of the recording and digitization, and also on the analysis window size and frame increment.

Next is an example using amplitude gating. The last 2 in the argument list stands for *f2* in the score.

```
asig pvadd ktime, 1, "oboe.pvoc", 1, 100, 2, 2, 0, 0, 2
```

Suppose the score for the above were to contain:

```
f2 0 512 7 0 256 1 256 1
```

Then those bins with amplitudes of 50% of the maximum or greater would be left unchanged, while those with amplitudes less than 50% of the maximum would be scaled down. In this case the lower the amplitude the more severe the scaling down would be. But suppose the score contains:

```
f2 0 512 5 1 512 .001
```

In this case lower amplitudes will be left unchanged and greater ones will be scaled down, turning the sound “upside-down” in terms of the amplitude spectrum! Functions can be arbitrarily complex. Just remember that the normalized amplitude values of the analysis are themselves the indices into the function.

Finally, both spectral extraction and amplitude gating can be used together. The example below will synthesize only those components that with a frequency deviation of less than 5Hz per frame and it will scale the amplitudes according to *f2*.

```
asig pvadd ktime, 1, "oboe.pvoc", 1, 100, 1, 1, 2, 5, 2
```



## USEFUL HINTS

By using several *pvadd* units together, one can gradually fade in different parts of the re-synthesis, creating various “filtering” effects. The author uses *pvadd* to synthesis one bin at a time to have control over each separate component of the re-synthesis.

If any combination of *ibins*, *ibinoffset*, and *ibinincr*, creates a situation where *pvadd* is asked to use a bin number greater than the number of bins in the analysis, it will just use all of the available bins, and give no complaint. So to use every bin just make *ibins* a big number (ie. 2000).

Expect to have to scale up the amplitudes by factors of 10-100, by the way.

Here is a complete example of the *pvadd* opcode. It uses the file *pvadd.csd* [examples/pvadd.csd]

### Example 729. Example of the *pvadd* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pvadd.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
Odbfs = 1
nchnls = 2

instr 1
; analyze "fox.wav" with PVANAL first
igatefn = p4
ktime line 0, p3, p3
asig pvadd ktime, 1, "fox.pvx", 1, 300, 2, 2, 0, 0, igatefn
outs asig*3, asig*3

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine wave
f 2 0 512 5 1 256 .001
f 3 0 512 7 0 256 1 256 1

i 1 0 2.8 2
i 1 + 2.8 3
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Richard Karpen  
Seattle, WA USA  
1998

New in Csound version 3.48, additional arguments version 3.56

# pvbufread

pvbufread — Reads from a phase vocoder analysis file and makes the retrieved data available.

## Description

*pvbufread* reads from a *pvoc* file and makes the retrieved data available to any following *pvinterp* and *pvcross* units that appear in an instrument before a subsequent *pvbufread* (just as *lpread* and *lpreson* work together). The data is passed internally and the unit has no output of its own.

## Syntax

```
pvbufread ktimepnt, ifile
```

## Initialization

*ifile* -- the *pvoc* number (n in *pvoc.n*) or the name in quotes of the analysis file made using *pvanal*. (See *pvoc*.)

## Performance

*ktimepnt* -- the passage of time, in seconds, through this file. *ktimepnt* must always be positive, but can move forwards or backwards in time, be stationary or discontinuous, as a pointer into the analysis file.

## Examples

The example below shows an example using *pvbufread* with *pvinterp* to interpolate between the sound of an oboe and the sound of a clarinet. The value of *kinterp* returned by a *linseg* is used to determine the timing of the transitions between the two sounds. The interpolation of frequencies and amplitudes are controlled by the same factor in this example, but for other effects it might be interesting to not have them synchronized in this way. In this example the sound will begin as a clarinet, transform into the oboe and then return again to the clarinet sound. The value of *kfreqscale2* is 1.065 because the oboe in this case is a semitone higher in pitch than the clarinet and this brings them approximately to the same pitch. The value of *kampscale2* is 0.75 because the analyzed clarinet was somewhat louder than the analyzed oboe. The setting of these two parameters make the transition quite smooth in this case, but such adjustments are by no means necessary or even advocated.

```
ktime1 line      0, p3, 3.5 ; used as index in the "oboe.pvoc" file
ktime2 line      0, p3, 4.5 ; used as index in the "clar.pvoc" file
kinterp linseg   1, p3*0.15, 1, p3*0.35, 0, p3*0.25, 0, p3*0.15, 1, p3*0.1, 1
pvbufread ktime1, "oboe.pvoc"
apv pvinterp ktime2,1,"clar.pvoc", 1, 1.065, 1, 0.75, 1-kinterp, 1-kinterp
```

Below is an example using *pvbufread* with *pvcross*. In this example the amplitudes used in the resynthesis gradually change from those of the oboe to those of the clarinet. The frequencies, of course, remain those of the clarinet throughout the process since *pvcross* does not use the frequency data from the file read by *pvbufread*.

```
ktime1 line      0, p3, 3.5 ; used as index in the "oboe.pvoc" file
ktime2 line      0, p3, 4.5 ; used as index in the "clar.pvoc" file
```

```
kcross expon 0.001, p3, 1
      pvbufread ktime1, "oboe.pvoc"
apv pvcross ktime2, 1, "clar.pvoc", 1-kcross, kcross
```

Here is a complete example of the pvbufread opcode. It uses the file *pvbufread.csd* [examples/pvbufread.csd]

### Example 730. Example of the pvbufread opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pvbufread.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 2

instr 1
; analyze "fox.wav" and "flute.aiff" with PVANAL first
ktime1 line 0, p3, .8 ; use a part of "flute.pvx" file
ktime2 line 0, p3, 1.2 ; use a part of "beats.pvx" file
kcross expon .03, p3, 1
      pvbufread ktime1, "flute.pvx"
asig pvcross ktime2, 1, "beats.pvx", 1-kcross, kcross
outs asig, asig

endin
</CsInstruments>
<CsScore>
i 1 0 3
i 1 + 10

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvcross*, *pvinterp*, *pvread*, *tableseg*, *tablexseg*

## Credits

Author: Richard Karpen  
Seattle, WA USA  
1997

# pvcross

`pvcross` — Applies the amplitudes from one phase vocoder analysis file to the data from a second file.

## Description

`pvcross` applies the amplitudes from one phase vocoder analysis file to the data from a second file and then performs the resynthesis. The data is passed, as described above, from a previously called `pvbufread` unit. The two k-rate amplitude arguments are used to scale the amplitudes of each files separately before they are added together and used in the resynthesis (see below for further explanation). The frequencies of the first file are not used at all in this process. This unit simply allows for cross-synthesis through the application of the amplitudes of the spectra of one signal to the frequencies of a second signal. Unlike `pvinterp`, `pvcross` does allow for the use of the `ispecwp` as in `pvoc` and `vpvoc`.

## Syntax

```
ares pvcross ktimepnt, kfmmod, ifile, kampscale1, kampscale2 [, ispecwp]
```

## Initialization

`ifile` -- the `pvoc` number (n in `pvoc.n`) or the name in quotes of the analysis file made using `pvanal`. (See `pvoc`.)

`ispecwp` (optional, default=0) -- if non-zero, attempts to preserve the spectral envelope while its frequency content is varied by `kfmmod`. The default value is zero.

## Performance

`ktimepnt` -- the passage of time, in seconds, through this file. `ktimepnt` must always be positive, but can move forwards or backwards in time, be stationary or discontinuous, as a pointer into the analysis file.

`kfmmod` -- a control-rate transposition factor: a value of 1 incurs no transposition, 1.5 transposes up a perfect fifth, and 0.5 down an octave.

`kampscale1`, `kampscale2` -- used to scale the amplitudes stored in each frame of the phase vocoder analysis file. `kampscale1` scale the amplitudes of the data from the file read by the previously called `pvbufread`. `kampscale2` scale the amplitudes of the file named by `ifile`.

By using these arguments, it is possible to adjust these values before applying the interpolation. For example, if file1 is much louder than file2, it might be desirable to scale down the amplitudes of file1 or scale up those of file2 before interpolating. Likewise one can adjust the frequencies of each to bring them more in accord with one another (or just the opposite, of course!) before the interpolation is performed.

## Examples

Below is an example using `pvbufread` with `pvcross`. In this example the amplitudes used in the resynthesis gradually change from those of the oboe to those of the clarinet. The frequencies, of course, remain those of the clarinet throughout the process since `pvcross` does not use the frequency data from the file read by `pvbufread`.

```

ktime1 line 0, p3, 3.5 ; used as index in the "oboe.pvoc" file
ktime2 line 0, p3, 4.5 ; used as index in the "clar.pvoc" file
kcross expon 0.001, p3, 1
pvbufread ktime1, "oboe.pvoc"
apv pvcross ktime2, 1, "clar.pvoc", 1-kcross, kcross

```

Here is a complete example of the pvcross opcode. It uses the file *pvcross.csd* [examples/pvcross.csd]

### Example 731. Example of the pvcross opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsoundOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o pvcross.wav -W ;; for file output any platform
</CsoundOptions>
<CsoundInstruments>

sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 2

instr 1
; analyze "beats.wav", "flute.aiff" and "mary.wav" with PVANAL first
ktime1 line 0, p3, 2 ; used as index in the "beats.pvx" file
ktime2 line 0, p3, 2.6 ; used as index in the "flute.pvx" or "mary.pvx"
pvbufread ktime1, "beats.pvx" ;take only amplitude from "beats.pvx"
if p4 = 0 then
asig pvcross ktime2, 1, "flute.pvx", 1, 0 ;and keep freqs of "flute.aiff"
asig = asig*.8 ;scale output
else
asig pvcross ktime2, 1, "mary.pvx", 1, 0 ;and keep freqs of "mary.wav"
asig = asig*.4 ;scale output
endif
outs asig, asig

endin
</CsoundInstruments>
<CsoundScore>
i 1 0 3 0
i 1 + 3 1

e
</CsoundScore>
</CsoundSynthesizer>

```

## See Also

*pvbufread*, *pvinterp*, *pvread*, *tableseg*, *tablexseg*

## Credits

Author: Richard Karpen  
 Seattle, Wash  
 1997

New in version 3.44

# pvinterp

pvinterp — Interpolates between the amplitudes and frequencies of two phase vocoder analysis files.

## Description

*pvinterp* interpolates between the amplitudes and frequencies, on a bin by bin basis, of two phase vocoder analysis files (one from a previously called *pvbufread* unit and the other from within its own argument list), allowing for user defined transitions between analyzed sounds. It also allows for general scaling of the amplitudes and frequencies of each file separately before the interpolated values are calculated and sent to the resynthesis routines. The *kfmod* argument in *pvinterp* performs its frequency scaling on the frequency values after their derivation from the separate scaling and subsequent interpolation is performed so that this acts as an overall scaling value of the new frequency components.

## Syntax

```
ares pvinterp ktimepnt, kfmod, ifile, kfreqscale1, kfreqscale2, \  
kampscale1, kampscale2, kfreqinterp, kampinterp
```

## Initialization

*ifile* -- the *pvoc* number (n in *pvoc.n*) or the name in quotes of the analysis file made using *pvanal*. (See *pvoc*.)

## Performance

*ktimepnt* -- the passage of time, in seconds, through this file. *ktimepnt* must always be positive, but can move forwards or backwards in time, be stationary or discontinuous, as a pointer into the analysis file.

*kfmod* -- a control-rate transposition factor: a value of 1 incurs no transposition, 1.5 transposes up a perfect fifth, and .5 down an octave.

*kfreqscale1*, *kfreqscale2*, *kampscale1*, *kampscale2* -- used in *pvinterp* to scale the frequencies and amplitudes stored in each frame of the phase vocoder analysis file. *kfreqscale1* and *kampscale1* scale the frequencies and amplitudes of the data from the file read by the previously called *pvbufread* (this data is passed internally to the *pvinterp* unit). *kfreqscale2* and *kampscale2* scale the frequencies and amplitudes of the file named by *ifile* in the *pvinterp* argument list and read within the *pvinterp* unit.

By using these arguments, it is possible to adjust these values before applying the interpolation. For example, if file1 is much louder than file2, it might be desirable to scale down the amplitudes of file1 or scale up those of file2 before interpolating. Likewise one can adjust the frequencies of each to bring them more in accord with one another (or just the opposite, of course!) before the interpolation is performed.

*kfreqinterp*, *kampinterp* -- used in *pvinterp*, determine the interpolation distance between the values of one phase vocoder file and the values of a second file. When the value of *kfreqinterp* is 1, the frequency values will be entirely those from the first file (read by the *pvbufread*), post scaling by the *kfreqscale1* argument. When the value of *kfreqinterp* is 0 the frequency values will be those of the second file (read by the *pvinterp* unit itself), post scaling by *kfreqscale2*. When *kfreqinterp* is between 0 and 1 the frequency values will be calculated, on a bin, by bin basis, as the percentage between each pair of frequencies (in other words, *kfreqinterp*=0.5 will cause the frequencies values to be half way between the values in the set of data from the first file and the set of data from the second file).

*kampinterp* works in the same way upon the amplitudes of the two files. Since these are k-rate arguments, the percentages can change over time making it possible to create many kinds of transitions between sounds.

## Examples

The example below shows an example using *pvbufread* with *pvinterp* to interpolate between the sound of an oboe and the sound of a clarinet. The value of *kinterp* returned by a *linseg* is used to determine the timing of the transitions between the two sounds. The interpolation of frequencies and amplitudes are controlled by the same factor in this example, but for other effects it might be interesting to not have them synchronized in this way. In this example the sound will begin as a clarinet, transform into the oboe and then return again to the clarinet sound. The value of *kfreqscale2* is 1.065 because the oboe in this case is a semitone higher in pitch than the clarinet and this brings them approximately to the same pitch. The value of *kampscale2* is 0.75 because the analyzed clarinet was somewhat louder than the analyzed oboe. The setting of these two parameters make the transition quite smooth in this case, but such adjustments are by no means necessary or even advocated.

```

ktime1 line 0, p3, 3.5 ; used as index in the "oboe.pvoc" file
ktime2 line 0, p3, 4.5 ; used as index in the "clar.pvoc" file
kinterp linseg 1, p3*0.15, 1, p3*0.35, 0, p3*0.25, 0, p3*0.15, 1, p3*0.1, 1
pvbufread ktime1, "oboe.pvoc"
apv pvinterp ktime2,1,"clar.pvoc", 1, 1.065, 1, 0.75, 1-kinterp, 1-kinterp

```

Here is a complete example of the *pvinterp* opcode. It uses the file *pvinterp.csd* [examples/pvinterp.csd]

### Example 732. Example of the *pvinterp* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pvinterp.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 2

instr 1
; analyze "fox.wav" and "flute.aiff" with PVANAL first
ktime1 line 0, p3, 2.8 ; used as index in the "fox.pvx" file
ktime2 line 0, p3, 3 ; used as index in the "flute.pvx" file
kinterp line 1, p3, 0
pvbufread ktime1, "fox.pvx"
asig pvinterp ktime2,1,"flute.pvx",.9, 3, .6, 1, kinterp,1-kinterp
outs asig, asig

endin
</CsInstruments>
<CsScore>
i 1 0 3
i 1 + 10

```

```
e  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*pvbufread, pvcross, pvread, tableseg, tablexseg*

## Credits

Author: Richard Karpen  
Seattle, Wash  
1997

## pvoc

pvoc — Implements signal reconstruction using an fft-based phase vocoder.

## Description

Implements signal reconstruction using an fft-based phase vocoder.

## Syntax

```
ares pvoc ktmpnt, kfmmod, ifilcod [, ispecwp] [, iextractmode] \  
      [, ifreqlim] [, igatefn]
```

## Initialization

*ifilcod* -- integer or character-string denoting a control-file derived from analysis of an audio signal. An integer denotes the suffix of a file *pvoc.m*; a character-string (in double quotes) gives a filename, optionally a full pathname. If not fullpath, the file is sought first in the current directory, then in the one given by the environment variable *SADIR* (if defined). *pvoc* control contains breakpoint amplitude and frequency envelope values organized for fft resynthesis. Memory usage depends on the size of the files involved, which are read and held entirely in memory during computation but are shared by multiple calls (see also *lpread*).

*ispecwp* (optional) -- if non-zero, attempts to preserve the spectral envelope while its frequency content is varied by *kfmmod*. The default value is zero.

*extractmode* (optional) -- determines if spectral extraction will be carried out and if so whether components that have changes in frequency below *ifreqlim* or above *ifreqlim* will be discarded. A value for *extractmode* of 1 will cause *pvoc* to synthesize only those components where the frequency difference between analysis frames is greater than *ifreqlim*. A value of 2 for *extractmode* will cause *pvoc* to synthesize only those components where the frequency difference between frames is less than *ifreqlim*. The default values for *extractmode* and *ifreqlim* are 0, in which case a simple resynthesis will be done. See examples under *pvadd* for how to use spectral extraction.

*igatefn* (optional) -- the number of a stored function which will be applied to the amplitudes of the analysis bins before resynthesis takes place. If *igatefn* is greater than 0 the amplitudes of each bin will be scaled by *igatefn* through a simple mapping process. First, the amplitudes of all of the bins in all of the frames in the entire analysis file are compared to determine the maximum amplitude value. This value is then used create normalized amplitudes as indices into the stored function *igatefn*. The maximum amplitude will map to the last point in the function. An amplitude of 0 will map to the first point in the function. Values between 0 and 1 will map accordingly to points along the function table. See examples under *pvadd* for how to use amplitude gating.

## Performance

*ktmpnt* -- The passage of time, in seconds, through the analysis file. *ktmpnt* must always be positive, but can move forwards or backwards in time, be stationary or discontinuous, as a pointer into the analysis file.

*kfmmod* -- a control-rate transposition factor: a value of 1 incurs no transposition, 1.5 transposes up a perfect fifth, and .5 down an octave.

*pvoc* implements signal reconstruction using an fft-based phase vocoder. The control data stems from a precomputed analysis file with a known frame rate.

This implementation of *pvoc* was originally written by Dan Ellis. It is based in part on the system of Mark Dolson, but the pre-analysis concept is new. The spectral extraction and amplitude gating (new in Csound version 3.56) were added by Richard Karpen based on functions in SoundHack by Tom Erbe.

## Examples

Here is an example of the *pvoc* opcode. It uses the file *pvoc.csd* [examples/pvoc.csd].

### Example 733. Example of the *pvoc* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pvoc.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 2

instr 1
; analyze "fox.wav" with PVANAL first
ispec = p4
ktime line 0, p3, 1.55
kfrq line .8, p3, 2
asig pvoc ktime, kfrq, "fox.pvx", ispec
outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 6 0
i 1 + 6 1 ;preserve spectral envelope
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*vpvoc*, *PVANAL*.

## Credits

Authors: Dan Ellis and Richard Karpen  
Seattle, Wash  
1997

# pvread

`pvread` — Reads from a phase vocoder analysis file and returns the frequency and amplitude from a single analysis channel or bin.

## Description

`pvread` reads from a `pvoc` file and returns the frequency and amplitude from a single analysis channel or bin. The returned values can be used anywhere else in the Csound instrument. For example, one can use them as arguments to an oscillator to synthesize a single component from an analyzed signal or a bank of `pvreads` can be used to resynthesize the analyzed sound using additive synthesis by passing the frequency and magnitude values to a bank of oscillators.

## Syntax

```
kfreq, kamp pvread ktimepnt, ifile, ibin
```

## Initialization

`ifile` -- the `pvoc` number (n in `pvoc.n`) or the name in quotes of the analysis file made using `pvanal`. (See `pvoc`.)

`ibin` -- the number of the analysis channel from which to return frequency in Hz and magnitude.

## Performance

`kfreq`, `kamp` -- outputs of the `pvread` unit. These values, retrieved from a phase vocoder analysis file, represent the values of frequency and amplitude from a single analysis channel specified in the `ibin` argument. Interpolation between analysis frames is performed at k-rate resolution and dependent of course upon the rate and direction of `ktimepnt`.

`ktimepnt` -- the passage of time, in seconds, through this file. `ktimepnt` must always be positive, but can move forwards or backwards in time, be stationary or discontinuous, as a pointer into the analysis file.

## Examples

The example below shows the use `pvread` to synthesize a single component at a time from a phase vocoder analysis file. It should be noted that the `kfreq` and `kamp` outputs can be used for any kind of synthesis, filtering, processing, and so on.

### Example 734. Example of the `pvread` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
```

```
; -o pvread.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 2

instr 1
; analyze "fox.wav" with PVANAL first
ibin = p4
ptime line 0, p3, 2.8
kfreq, kamp pvread ptime, "fox.pvx", ibin ;read data from 7th analysis bin.
asig poscil kamp, kfreq, 1 ;function 1 is a stored sine
outs asig*5, asig*5 ;compensate loss of volume

endin
</CsInstruments>
<CsScore>
;sine wave
f1 0 4096 10 1

i 1 0 6 7
i 1 + 6 15
i 1 + 2 25
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvbufread*, *pvcross*, *pvinterp*, *tableseg*, *tablexseg*

## Credits

Author: Richard Karpen  
Seattle, Wash  
1997

New in version 3.44

# pvsadsyn

pvsadsyn — Resynthesize using a fast oscillator-bank.

## Description

Resynthesize using a fast oscillator-bank.

## Syntax

```
ares pvsadsyn fsrc, inoscs, kfmod [, ibinoffset] [, ibinincr] [, iinit]
```

## Initialization

*inoscs* -- The number of analysis bins to synthesise. Cannot be larger than the size of *fsrc* (see *pvsinfo*), e.g. as created by *pvsanal*. Processing time is directly proportional to *inoscs*.

*ibinoffset* (optional, default=0) -- The first (lowest) bin to resynthesise, counting from 0 (default = 0).

*ibinincr* (optional) -- Starting from bin *ibinoffset*, resynthesize bins *ibinincr* apart.

*iinit* (optional) -- Skip reinitialization. This is not currently implemented for any of these opcodes, and it remains to be seen if it is even practical.

## Performance

*kfmod* -- Scale all frequencies by factor *kfmod*. 1.0 = no change, 2 = up one octave.

*pvsadsyn* is experimental, and implements the oscillator bank using a fast direct calculation method, rather than a lookup table. This takes advantage of the fact, empirically arrived at, that for the analysis rates generally used, (and presuming analysis using *pvsanal*, where frequencies in a bin change only slightly between frames) it is not necessary to interpolate frequencies between frames, only amplitudes. Accurate resynthesis is often contingent on the use of *pvsanal* with *iwinsize* = *ifftsize*\*2.

This opcode is the most likely to change, or be much extended, according to feedback and advice from users. It is likely that a full interpolating table-based method will be added, via a further optional *iarg*. The parameter list to *pvsadsyn* mimics that for *pvadd*, but excludes spectral extraction.

## Examples

Here is an example of the *pvsadsyn* opcode. It uses the file *pvsadsyn.csd* [examples/pvsadsyn.csd].

### Example 735. Example of the pvsadsyn opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
; Audio out  Audio in
```

```

-odac      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o pvsadsyn.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 16
nchnls = 1
0dbfs = 1

;; example written by joachim heintz 2009

opcode FileToPvsBuf, iik, Siiii
;writes an audio file at the first k-cycle to a fft-buffer (via pvsbuffer)
Sfile, ifftsize, ioverlap, iwinshape, iwinshape xin
ktimek timeinstk
if ktimek == 1 then
ilen filelen Sfile
kcycles = ilen * kr; number of k-cycles to write the fft-buffer
kcount init 0
loop:
ain soundin Sfile
fftin pvsanal ain, ifftsize, ioverlap, iwinshape, iwinshape
ibuf, ktim pvsbuffer fftin, ilen + (ifftsize / sr)
loop_lt kcount, 1, kcycles, loop
xout ibuf, ilen, ktim
endif
endop

instr 1
istretch = p4; time stretching factor
ifftsize = 1024
ioverlap = ifftsize / 4
iwinshape = 1; von-Hann window
ibuffer, ilen, k0 FileToPvsBuf "fox.wav", ifftsize, ioverlap, iwinshape, iwinshape
p3 = istretch * ilen; set p3 to the correct value
ktmpnt linseg 0, p3, ilen; time pointer
fread pvsbufread ktmpnt, ibuffer; read the buffer
aout pvsadsyn fread, 10, 1; resynthesis with the first 10 bins
out aout
endin

</CsInstruments>
<CsScore>
i 1 0 1 20
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*pvsanal*, *pvsynth*, *pvsadsyn*

## Credits

Author: Richard Dobson  
August 2001

New in version 4.13

# pvsanal

pvsanal — Generate an fsig from a mono audio source ain, using phase vocoder overlap-add analysis.

## Description

Generate an fsig from a mono audio source ain, using phase vocoder overlap-add analysis.

## Syntax

```
fsig pvsanal ain, ifftsize, ioverlap, iwinsize, iwintype [, iformat] [, iinit]
```

## Initialization

*ifftsize* -- The FFT size in samples. Need not be a power of two (though these are especially efficient), but must be even. Odd numbers are rounded up internally. *ifftsize* determines the number of analysis bins in *fsig*, as  $ifftsize/2 + 1$ . For example, where *ifftsize* = 1024, *fsig* will contain 513 analysis bins, ordered linearly from the fundamental to Nyquist. The fundamental of analysis (which in principle gives the lowest resolvable frequency) is determined as  $sr/ifftsize$ . Thus, for the example just given and assuming  $sr = 44100$ , the fundamental of analysis is 43.07Hz. In practice, due to the phase-preserving nature of the phase vocoder, the frequency of any bin can deviate bilaterally, so that DC components are recorded. Given a strongly pitched signal, frequencies in adjacent bins can bunch very closely together, around partials in the source, and the lowest bins may even have negative frequencies.

As a rule, the only reason to use a non power-of-two value for *ifftsize* would be to match the known fundamental frequency of a strongly pitched source. Values with many small factors can be almost as efficient as power-of-two sizes; for example: 384, for a source pitched at around low A=110Hz.

*ioverlap* -- The distance in samples (“hop size”) between overlapping analysis frames. As a rule, this needs to be at least  $ifftsize/4$ , e.g. 256 for the example above. *ioverlap* determines the underlying analysis rate, as  $sr/ioverlap$ . *ioverlap* does not require to be a simple factor of *ifftsize*; for example a value of 160 would be legal. The choice of *ioverlap* may be dictated by the degree of pitch modification applied to the *fsig*, if any. As a rule of thumb, the more extreme the pitch shift, the higher the analysis rate needs to be, and hence the smaller the value for *ioverlap*. A higher analysis rate can also be advantageous with broadband transient sounds, such as drums (where a small analysis window gives less smearing, but more frequency-related errors).

Note that it is possible, and reasonable, to have distinct fsigs in an orchestra (even in the same instrument), running at different analysis rates. Interactions between such fsigs is currently unsupported, and the fsig assignment opcode does not allow copying between fsigs with different properties, even if the only difference is in *ioverlap*. However, this is not a closed issue, as it is possible in theory to achieve crude rate conversion (especially with regard to in-memory analysis files) in ways analogous to time-domain techniques.

*iwinsize* -- The size in samples of the analysis window filter (as set by *iwintype*). This must be at least *ifftsize*, and can usefully be larger. Though other proportions are permitted, it is recommended that *iwinsize* always be an integral multiple of *ifftsize*, e.g. 2048 for the example above. Internally, the analysis window (Hamming, von Hann) is multiplied by a sinc function, so that amplitudes are zero at the boundaries between frames. The larger analysis window size has been found to be especially important for oscillator bank resynthesis (e.g. using *pvsadsyn*), as it has the effect of increasing the frequency resolution of the analysis, and hence the accuracy of the resynthesis. As noted above, *iwinsize* determines the overall latency of the analysis/resynthesis system. In many cases, and especially in the absence of pitch modifications, it will be found that setting  $iwinsize=ifftsize$  works very well, and offers the lowest latency.

*iwintype* -- The shape of the analysis window. Currently three choices computed windows implemented:

- 0 = Hamming window
- 1 = von Hann window
- 3 = Kaiser window (not in sliding form)

These are also supported by the PVOC-EX file format. The window type is stored as an internal attribute of the *fsig*, together with the other parameters (see *pvsinfo*). Other types may be implemented later on; if the value of *wintype* is strictly negative then the absolute value is used as the number of an *f*-table which must pre-exist. A significant issue here is the common constraint of *f*-tables to power-of-two sizes, so this method does not offer a complete solution. Most users will find the Hamming window meets all normal needs, and can be regarded as the default choice.

*iformat* -- (optional) The analysis format. Currently only one format is implemented by this opcode:

- 0 = amplitude + frequency

This is the classic phase vocoder format; easy to process, and a natural format for oscillator-bank resynthesis. It would be very easy (tempting, one might say) to treat an *fsig* frame not purely as a phase vocoder frame but as a generic additive synthesis frame. It is indeed possible to use an *fsig* this way, but it is important to bear in mind that the two are not, strictly speaking, directly equivalent.

Other important formats (supported by PVOC-EX) are:

- 1 = amplitude + phase
- 2 = complex (real + imaginary)

*iformat* is provided in case it proves useful later to add support for these other formats. Formats 0 and 1 are very closely related (as the phase is “wrapped” in both cases - it is a trivial matter to convert from one to the other), but the complex format might warrant a second explicit signal type (a “*csig*”) specifically for convolution-based processes, and other processes where the full complement of arithmetic operators may be useful.

*iinit* -- (optional) Skip reinitialization. This is not currently implemented for any of these opcodes, and it remains to be seen if it is even practical.



### Warning

It is unsafe to use the same *f*-variable for both input and output of *pvs* opcodes. Using the same one might lead to undefined behavior on some opcodes. Use a different one on the left and right sides of the opcode.

## Examples

Here is an example of the *pvsanal* opcode. It uses the file *pvsanal.csd* [examples/*pvsanal.csd*].

### Example 736. Example of the *pvsanal* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
```

```
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pvsanal.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;pvsanal has no influence when there is no transformation of original sound

ifftsize = p4
ioverlap = ifftsize / 4
iwinsize = ifftsize
iwinshape = 1 ;von-Hann window
Sfile = "fox.wav"
ain soundin Sfile
fftin pvsanal ain, ifftsize, ioverlap, iwinsize, iwinshape ;fft-analysis of the audio-signal
fftblur pvscale fftin, p5 ;scale
aout pvsynth fftblur ;resynthesis
outs aout, aout

endin

</CsInstruments>
<CsScore>
s
i 1 0 3 512 1 ;original sound - ifftsize of pvsanal does not have any influence
i 1 3 3 1024 1 ;even with different
i 1 6 3 2048 1 ;settings

s
i 1 0 3 512 1.5 ;but transformation - here a fifth higher
i 1 3 3 1024 1.5 ;but with different settings
i 1 6 3 2048 1.5 ;for ifftsize of pvsanal

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Richard Dobson  
August 2001

New in version 4.13

# pvsarp

pvsarp — Arpeggiate the spectral components of a streaming pv signal.

## Description

This opcode arpeggiates spectral components, by amplifying one bin and attenuating all the others around it. Used with an LFO it will provide a spectral arpeggiator similar to Trevor Wishart's CDP program `specarp`.

## Syntax

```
fsig pvsarp fsigin, kbin, kdepth, kgain
```

## Performance

*fsig* -- output pv stream

*fsigin* -- input pv stream

*kbin* -- target bin, normalised 0 - 1 (0Hz - Nyquist).

*kdepth* -- depth of attenuation of surrounding bins

*kgain* -- gain boost applied to target bin



### Warning

It is unsafe to use the same f-variable for both input and output of pvs opcodes. Using the same one might lead to undefined behavior on some opcodes. Use a different one on the left and right sides of the opcode.

## Examples

Here is an example of the pvsarp opcode. It uses the file `pvsarp.csd` [examples/pvsarp.csd]

### Example 737. Example of the pvsarp opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      ;;-d      RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o pvsarp.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
```

```

ksmps = 100
nchnls = 1
0dbfs = 1

instr 1
asig in ; get the signal in
idepth = p4

fsig pvsanal asig, 1024, 256, 1024, 1 ; analyse it
kbin oscili 0.1, 0.5, 1 ; ftable 1 in the 0-1 range
ftps pvsarp fsig, kbin+0.01, idepth, 2 ; arpeggiate it (range 220.5 - 2425.5)
atps pvsynth ftps ; synthesise it

out atps
endin

</CsInstruments>
<CsScore>
f 1 0 4096 10 1 ;sine wave

i 1 0 10 0.9
i 1 + 10 0.5
e

</CsScore>
</CsoundSynthesizer>

```

Here is another example of the pvsarp opcode. It uses the file *pvsarp2.csd* [examples/pvsarp2.csd]

### Example 738. Example of the pvsarp opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out Audio in
-odac ;;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o pvsarp2.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 16
nchnls = 1
0dbfs = 1

;; example written by joachim heintz 2009

instr 1
ifftsize = 1024
ioverlap = ifftsize / 4
iwinshape = ifftsize
iwinshape = 1; von-Hann window
Sfile1 = "fox.wav"
ain1 soundin Sfile1
fftin pvsanal ain1, ifftsize, ioverlap, iwinshape, iwinshape
;make 3 independently moving accentuations in the spectrum
kbin1 linseg 0.05, p3/2, .05, p3/2, .05
farp1 pvsarp fftin, kbin1, .9, 10
kbin2 linseg 0.075, p3/2, .1, p3/2, .075

```

```
farf2 pvsarp fftin, kbin2, .9, 10
kbin3 linseg 0.02, p3/2, .03, p3/2, .04
farf3 pvsarp fftin, kbin3, .9, 10
;resynthesize and add them
aout1 pvsynth farf1
aout2 pvsynth farf2
aout3 pvsynth farf3
aout = aout1*.3 + aout2*.3 + aout3*.3
out aout
endin

</CsInstruments>
<CsScore>
i 1 0 3
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvsanal, pvsynth, pvsadsyn*

## Credits

Author: Victor Lazzarini  
April 2005

New plugin in version 5

April 2005.

# pvsbandp

pvsbandp — A band pass filter working in the spectral domain.

## Description

Filter the pvoc frames, passing bins whose frequency is within a band, and with linear interpolation for transitional bands.

## Syntax

```
fsig pvsbandp fsigin, xlowcut, xlowfull, \
      xhighfull, xhighcut[, ktype]
```

## Performance

*fsig* -- output pv stream

*fsigin* -- input pv stream.

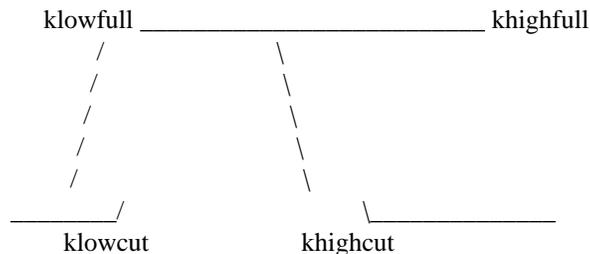
*xlowcut*, *xlowfull*, *xhighfull*, *xhighcut* -- define a trapezium shape for the band that is passed. The a-rate versions only apply to the sliding case.

*ktype* -- specifies the shape of the transitional band. If at the default value of zero the shape is as below, with linear transition in amplitude. Other values yield an exponential shape:

$$(1 - \exp(-r \cdot \text{type})) / (1 - \exp(-\text{type}))$$

This includes a linear dB shape when *ktype* is  $\log(10)$  or about 2.30.

The opcode performs a band-pass filter with a spectral envelope shaped like



## Examples

Here is an example of the pvsbandp opcode. It uses the file *pvsbandp.csd* [examples/pvsbandp.csd].

### Example 739. Example of the pvsbandp opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o pvsbandp.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 16
nchnls = 1
0dbfs = 1

;; example written by joachim heintz 2009

instr 1
Sfile = "fox.wav"
klowcut = 100
klowfull = 200
khighfull = 1900
khighcut = 2000
ain soundin Sfile
fftin pvsanal ain, 1024, 256, 1024, 1; fft-analysis of the audio-signal
fftbp pvsbandp fftin, klowcut, klowfull, khighfull, khighcut ; band pass
abp pvsynth fftbp; resynthesis
out abp
endin

</CsInstruments>
<CsScore>
i 1 0 3
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvsanal, pvsynth, pvsadsyn, pvsbandr*

## Credits

Author: John ffitth  
December 2007

# pvsbandr

pvsbandr — A band reject filter working in the spectral domain.

## Description

Filter the pvoc frames, rejecting bins whose frequency is within a band, and with linear interpolation for transitional bands.

## Syntax

```
fsig pvsbandr fsigin, xlowcut, xlowfull, \
      xhighfull, xhighcut[, ktype]
```

## Performance

*fsig* -- output pv stream

*fsigin* -- input pv stream.

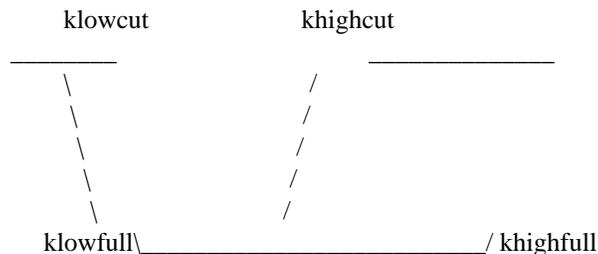
*xlowcut*, *xlowfull*, *xhighfull*, *xhighcut* -- define a trapezium shape for the band that is rejected. The a-rate versions only apply to the sliding case.

*ktype* -- specifies the shape of the transitional band. If at the default value of zero the shape is as below, with linear transition in amplitude. Other values give an exponential curve

$$(1 - \exp(r * \text{type})) / (1 - \exp(\text{type}))$$

This includes a linear dB shape when *ktype* is  $\log(10)$  or about 2.30.

The opcode performs a band-reject filter with a spectral envelope shaped like



## Examples

Here is an example of the pvsbandr opcode. It uses the file *pvsbandr.csd* [examples/pvsbandr.csd].

### Example 740. Example of the pvsbandr opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o pvsbandr.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 16
nchnls = 1
0dbfs = 1

;; example written by joachim heintz 2009

instr 1
Sfile = "fox.wav"
klowcut = 100
klowfull = 200
khighfull = 1900
khighcut = 2000
ain soundin Sfile
fftin pvsanal ain, 1024, 256, 1024, 1; fft-analysis of the audio-signal
fftbp pvsbandr fftin, klowcut, klowfull, khighfull, khighcut ; band reject
abp pvsynth fftbp; resynthesis
out abp
endin

</CsInstruments>
<CsScore>
i 1 0 3
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvsanal, pvsynth, pvsadsyn, pvsbandp*

## Credits

Author: John ffitc  
December 2007

# pvsbin

pvsbin — Obtain the amp and freq values off a PVS signal bin.

## Description

Obtain the amp and freq values off a PVS signal bin as k-rate variables.

## Syntax

```
kamp, kfr pvsbin fsig, kbin
```

## Performance

*kamp* -- bin amplitude

*kfr* -- bin frequency

*fsig* -- an input pv stream

*kbin* -- bin number

## Examples

Here is an example of the pvsbin opcode. It uses the file *pvsbin.csd* [examples/pvsbin.csd]. This example uses realtime input, but you can also use it for soundfile input.

### Example 741. Example of the pvsbin opcode

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o pvsbin.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 1
ifftsize = 1024
iwtype = 1 /* cleaner with hanning window */

a1 soundin "input.wav" ;select a soundifile
a1 inch 1 ;Use realtime input

fsig pvsanal a1, ifftsize, ifftsize/4, ifftsize, iwtype
```

```
kamp, kfr pvsbin fsig, 10
adm oscil kamp, kfr, 1

    out adm
endin

</CsInstruments>
<CsScore>
; sine wave
f 1 0 4096 10 1

i 1 0 30
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvsanal, pvsynth, pvsadsyn*

## Credits

Author: Victor Lazzarini  
August 2006

# pvsblur

pvsblur — Average the amp/freq time functions of each analysis channel for a specified time.

## Description

Average the amp/freq time functions of each analysis channel for a specified time (truncated to number of frames). As a side-effect the input pvoc stream will be delayed by that amount.

## Syntax

```
fsig pvsblur fsigin, kblurtime, imaxdel
```

## Performance

*fsig* -- output pv stream

*fsigin* -- input pv stream.

*kblurtime* -- time in secs during which windows will be averaged .

*imaxdel* -- maximum delay time, used for allocating memory used in the averaging operation.

This opcode will blur a pvstream by smoothing the amplitude and frequency time functions (a type of low-pass filtering); the amount of blur will depend on the length of the averaging period, larger blur times will result in a more pronounced effect.



### Warning

It is unsafe to use the same f-variable for both input and output of pvs opcodes. Using the same one might lead to undefined behavior on some opcodes. Use a different one on the left and right sides of the opcode.

## Examples

Here is an example of the use of the *pvsblur* opcode. It uses the file *pvsblur.csd* [examples/pvsblur.csd].

### Example 742. Example of the *pvsblur* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 16
nchnls = 1
0dbfs = 1
```

```
;; example written by joachim heintz 2009

instr 1
ifftsize = 1024
ioverlap = ifftsize / 4
iwinsize = ifftsize
iwinshape = 1; von-Hann window
Sfile = "fox.wav"
ain soundin Sfile
fftin pvsanal ain, ifftsize, ioverlap, iwinsize, iwinshape; fft-analysis of the audio-signal
fftblur pvsblur fftin, p4, 1; blur
aout pvsynth fftblur; resynthesis
out aout
endin

</CsInstruments>
<CsScore>
i 1 0 3 0
i 1 3 3 .1
i 1 6 3 .5
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvsanal, pvsynth, pvsadsyn*

## Credits

Author: Victor Lazzarini  
November 2004

New plugin in version 5

November 2004.

# pvsbuffer

`pvsbuffer` — This opcode creates and writes to a circular buffer for f-signals (streaming PV signals).

## Description

This opcode sets up and writes to a circular buffer of length `ilen` (secs), giving a handle for the buffer and a time pointer, which holds the current write position (also in seconds). It can be used with one or more `pvsbufread` opcodes. Writing is circular, wrapping around at the end of the buffer.

## Syntax

```
ihandle, ktime pvsbuffer fsig, ilen
```

## Initialisation

### Initialisation

`ihandle` -- handle identifying this particular buffer, which should be passed to a reader opcode.

`ilen` -- buffer length in seconds.

`fsig` -- an input pv stream

`ktime` -- the current time of writing in the buffer

`pvsbuffer` stores `fsig` in a buffer which can be read by `pvsbufread` using the handle `ihandle`. Different buffers will have different handles so they can be read independently by different `pvsbufread` opcodes. `pvsbuffer` gives in its output the current time (`ktime`) inside the ring buffer which has just been written.

## Examples

See `pvsbufread` for examples of the `pvsbuffer` opcode.

## See also

`pvsbufread`

## Credits

Author: Victor Lazzarini  
July 2007

# pvsbufread

`pvsbufread` — This opcode reads a circular buffer of f-signals (streaming PV signals).

## Description

This opcode reads from a circular buffer of length `ilen` (secs), taking a handle for the buffer and a time pointer, which holds the current read position (also in seconds). It is used in conjunction with a *pvsbuffer* opcode. Reading is circular, wrapping around at the end of the buffer.

## Syntax

```
fsg pvsbufread ktime, khandle[, ilo, ihi, iclear]
```

## Initialisation

*ilo, ihi* -- set the lowest and highest freqs to be read from the buffer (defaults to 0, Nyquist).

*iclear* -- set to 1 to clear output fsg before every write (default 1), 0 tells the opcode not to clear the output fsg. This is relevant when writing to subsets of an fsg frame using *ilo, ihi*.

## Performance

*fsg* -- output pv stream

*ktime* -- time position of reading pointer (in secs).

*khandle* -- handle identifying the buffer to be read. When using k-rate handles, it is important to initialise the k-rate variable to a given existing handle. When changing buffers, fsg buffers need to be compatible (same fsg format).

With this opcode and *pvsbuffer*, it is possible to, among other things:

- time-stretch/compress a fsg stream, by reading it at different rates
- delay a fsg or portions of it.
- 'brassage' two or more fsigs by switching buffers, since the reading handles are k-rate. Note that, when using k-rate handles, it is important to initialise the k-rate variable to a given handle (so that the fsg initialisation can take place) and it is only possible to switch handles between compatible fsg buffers (with the same fftsize and overlap).



### Note

It is important that the handle value passed to *pvsbufread* is valid and was created by *pvsbuffer*. Csound will crash with invalid handles.

## Examples

Here is an example of the *pvsbufread* opcode. It does 'brassage' by switching between two buffers.

**Example 743. Example of the pvbufread opcode**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

fsig1      pvsanal      asig1, 1024, 256, 1024, 1
fsig2      pvsanal      asig2, 1024, 256, 1024, 1

ibuf1, kt1 pvsbuffer    fsig1, 10      ; 10-sec buf with fsig1
ibuf2, kt2 pvsbuffer    fsig2, 7       ; 7-sec buf with fsig2

khan        init        ibuf1          ; initialise handle to buf1

if ktrig > 0 then                ; switch buffers according to trigger
khan = ibuf2
else
khan = ibuf1
endif

fsb          pvbufread  kt1, khan      ; read buffer

```

Here is an example of the pvbufread opcode. It uses the file *pvbufread.csd* [examples/pvbufread.csd].

**Example 744. Example of the pvbufread opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o pvbufread.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 16
nchnls = 1
0dbfs = 1

;; example written by joachim heintz 2009

opcode FileToPvsBuf, iik, Siiii
;;writes an audio file at the first k-cycle to a fft-buffer (via pvsbuffer)
Sfile, ifftsize, ioverlap, iwinshape, iwinshape xin
ktimek timeinstk
if ktimek == 1 then
ilen filelen Sfile
kcycles = ilen * kr; number of k-cycles to write the fft-buffer
kcount init 0
loop:
ain soundin Sfile
fftin pvsanal ain, ifftsize, ioverlap, iwinshape, iwinshape
ibuf, ktim pvsbuffer fftin, ilen + (ifftsize / sr)
loop_lt kcount, 1, kcycles, loop
xout ibuf, ilen, ktim
endif
endop

```

```
instr 1
  ifftsize = 1024
  ioverlap = ifftsize / 4
  iwinsize = ifftsize
  iwinshape = 1; von-Hann window
  ibuffer, ilen, k0 FileToPvsBuf "fox.wav", ifftsize, ioverlap, iwinsize, iwinshape
  ktmpnt linseg ilen, p3, 0; reads the buffer backwards in p3 seconds
  fread pvsbufread ktmpnt, ibuffer
  aout pvsynth fread
  out aout
endin

</CsInstruments>
<CsScore>
i 1 0 5
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvsanal, pvsynth, pvsbuffer, pvsadsyn*

## Credits

Author: Victor Lazzarini  
July 2007

# pvsbufread2

*pvsbufread2* — This opcode reads a circular buffer of f-signals (streaming PV signals), with binwise additional delays.

## Description

This opcode reads from a circular buffer of length *ilen* (secs), taking a handle for the buffer and a time pointer, which holds the current read position (also in seconds). It is used in conjunction with a *pvsbuffer* opcode. Reading is circular, wrapping around at the end of the buffer. Extra delay times are taken from a function table, with each point on it defining a delay time in seconds affecting the corresponding bin.

## Syntax

```
fsig pvsbufread2 ktime, khandle, ift1, ift2
```

## Initialization

*ift1* -- function table with at least  $\text{fftsize}/2+1$  points where delays (in secs) for bin amplitudes are set (function table positions are equivalent to bin numbers)

*ift2* -- function table with at least  $\text{fftsize}/2+1$  points where delays (in secs) for bin frequencies are set (function table positions are equivalent to bin numbers)

## Performance

*fsig* -- output pv stream

*ktime* -- time position of reading pointer (in secs).

*khandle* -- handle identifying the buffer to be read. When using k-rate handles, it is important to initialise the k-rate variable to a given existing handle. When changing buffers, *fsig* buffers need to be compatible (same *fsig* format).

With this opcode and *pvsbuffer*, it is possible to, among other things:

- time-stretch/compress a *fsig* stream, by reading it at different rates
- delay a *fsig* or portions of it.
- 'brassage' two or more *fsigs* by switching buffers, since the reading handles are k-rate. Note that, when using k-rate handles, it is important to initialise the k-rate variable to a given handle (so that the *fsig* initialisation can take place) and it is only possible to switch handles between compatible *fsig* buffers (with the same *fftsize* and *overlap*).



### Note

It is important that the handle value passed to *pvsbufread2* is valid and was created by *pvsbuffer*. Csound will crash with invalid handles.

## Examples

Here is an example of the *pvsbufread2* opcode. It uses the file *pvsbufread2.csd* [examples/pvsbufread2.csd].

**Example 745. Example of the pvsbufread2 opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>

<CsInstruments>
ksmps = 64
0dbfs = 1
nchnls = 2

instr 1
kcnt   init      0
ifftsize =      2048
ihop   =         ifftsize/4

a1     diskin2   "beats.wav", 1, 0, 1
a1     =         a1*0.5
fsig1  pvsanal   a1, ifftsize, ihop, ifftsize, 1
ih, kt pvsbuffer fsig1, 10

fsig2  pvsbufread2 kt, ih, 1, 1
fsig3  pvsbufread2 kt, ih, 2, 2

a2     pvsynth   fsig3
a3     pvsynth   fsig2

      outs      a2, a3

endin
</CsInstruments>

<CsScore>
f1 0 2048 -7 0 128 1.1 128 0.5 256 1.8 512 1.1 1024 0.1
f2 0 2048 -7 1 128 0.2 128 0.05 256 0.5 512 0.9 1024 0.1

i1 0 60
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvsanal*, *pvsynth*, *pvsbuffer*, *pvsadsyn*

## Credits

Author: Victor Lazzarini  
July 2011

# pvscale

pvscale — Scale the frequency components of a pv stream.

## Description

Scale the frequency components of a pv stream, resulting in pitch shift. Output amplitudes can be optionally modified in order to attempt formant preservation.

## Syntax

```
fsig pvscale fsignin, kscal[, kkeepform, kgain, kcoefs]
```

## Performance

*fsig* -- output pv stream

*fsignin* -- input pv stream

*kscal* -- scaling ratio.

*kkeepform* -- attempt to keep input signal formants; 0: do not keep formants; 1: keep formants using a liftered cepstrum method; 2: keep formants by using a true envelope method (defaults to 0).

*kgain* -- amplitude scaling (defaults to 1).

*kcoefs* -- number of cepstrum coefs used in formant preservation (defaults to 80).

The quality of the pitch shift will be improved with the use of a Hanning window in the pvoc analysis. Formant preservation method 1 is less intensive than method 2, which might not be suited to realtime use.



### Warning

It is unsafe to use the same f-variable for both input and output of pvs opcodes. Using the same one might lead to undefined behavior on some opcodes. Use a different one on the left and right sides of the opcode.

## Examples

### Example 746. Example

```
asig in ; get the signal in

fsig pvsanal asig, 1024, 256, 1024, 1 ; analyse it
ftps pvscale fsig, 1.5, 1, 1 ; transpose it keeping formants
atps pvsynth ftps ; synthesise it

adp delay 0.1 ; delay original signal
adel deltapn 1024 ; by 1024 samples
delayw asig
out atps + adel ; add tranposed and original
```

The example above shows a vocal harmoniser. The delay is necessary to time-align the signals, as the analysis-synthesis process will imply a delay of 1024 samples between the analysis input and the synthesis output.

Here is an example of the use of the *pvscale* opcode. It uses the file *pvscale.csd* [examples/pvscale.csd].

### Example 747. Example of the *pvscale* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 16
nchnls = 1
0dbfs = 1

; example written by joachim heintz 2009

instr 1
ifftsize = 1024
ioverlap = ifftsize / 4
iwinshape = 1; von-Hann window
Sfile = "fox.wav"
ain soundin Sfile
fftin pvsanal ain, ifftsize, ioverlap, iwinshape; fft-analysis of the audio-signal
fftblur pvscale fftin, p4, p5, p6; scale
aout pvsynth fftblur; resynthesis
out aout
endin

</CsInstruments>
<CsScore>
i 1 0 3 1 0 1; original sound
i 1 3 3 1.5 0 2; fifth higher without ...
i 1 6 3 1.5 1 2; ... and with different ...
i 1 9 3 1.5 2 5; ... kinds of formant preservation
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvsanal*, *pvsynth*, *pvsadsyn*

## Credits

Author: Victor Lazzarini  
November 2004

New plugin in version 5

November 2004.

# pvscent

pvscent — Calculate the spectral centroid of a signal.

## Description

Calculate the spectral centroid of a signal from its discrete Fourier transform.

## Syntax

```
kcent pvscent fsig
```

## Performance

*kcent* -- the spectral centroid

*fsig* -- an input pv stream

## Examples

Here is an example of the use of the *pvscent* opcode. It uses the file *pvscent.csd* [examples/pvscent.csd].

### Example 748. Example of the *pvscent* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 16
nchnls = 1
0dbfs = 1

;; example written by joachim heintz 2009

giSine ftgen 0, 0, 4096, 10, 1

instr 1
irefrtm = p4; time for generating new values for the spectral centroid
ifftsize = 1024
ioverlap = ifftsize / 4
iwinshape = ifftsize
iwinshape = 1; von-Hann window
;Sfile = "flute-C-octave0.wav"
Sfile = "fox.wav"
ain soundin Sfile
fftin pvsanal ain, ifftsize, ioverlap, iwinshape, iwinshape; fft-analysis of the audio-signal
ktrig metro 1 / irefrtm
if ktrig == 1 then
kcenter pvscent fftin; spectral center
endif
```

```
aout oscil .2, kcenter, giSine
out aout
endin

</CsInstruments>
<CsScore>
i 1 0 2.757 .3
i 1 3 2.757 .05
i 1 6 2.757 .005
i 1 9 2.757 .001
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvsanal, pvsynth, pvsadsyn, pvspitch*

## Credits

Author: John ffitch  
March 2005

New plugin in version 5

March 2005.

# pvsceps

`pvsceps` — Calculate the cepstrum of a pvs input, optionally liftering coefficients.

## Description

## Syntax

```
keps[] pvsceps fsig[, icoefs]
```

## Initialization

`icoefs` -- the number of retained coefficients in the cepstrum output. By default, no coefficients are liftered.

## Performance

`keps[]` -- the cepstrum output, an array of size  $N/2+1$ , where  $N$  is equivalent to the FFT size of the input `fsig`.

`fsig` -- an input pv stream

## Examples

Here is an example of the use of the `pvsceps` opcode. It uses the file `pvsceps.csd` [examples/pvsceps.csd].

### Example 749. Example of the `pvsceps` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>

instr 1

a1 diskin "fox.wav",1,0,1
k1 randh 80, 2.5
a2 vco2 8, 220+k1
fsig pvsanal a1,1024,256,1024,1
fsig2 pvsanal a2,1024,256,1024,1
keps[] pvsceps fsig,30
kenv[] cepsinv keps
fenv tab2pvs r2c(kenv)
fvoc pvsfilter fsig2, fenv, 1
asig pvsynth fvoc

    out asig
endin

</CsInstruments>
<CsScore>
i1 0 60
```

```
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*pvsanal, pvsynth, pvsadsyn, pvspitch*

## Credits

Author: Victor Lazzarini  
2014

# pvscross

pvscross — Performs cross-synthesis between two source fsigs.

## Description

Performs cross-synthesis between two source fsigs.

## Syntax

```
fsig pvscross fsrc, fdest, kamp1, kamp2
```

## Performance

The operation of this opcode is identical to that of *pvcross* (q.v.), except in using *fsigs* rather than analysis files, and the absence of spectral envelope preservation. The amplitudes from *fsrc* and *fdest* (using scale factors *kamp1* for *fsrc* and *kamp2* for *fdest*) are applied to the frequencies of *fsrc*. *kamp1* and *kamp2* must not exceed the range 0 to 1.

With this opcode, cross-synthesis can be performed on real-time audio input, by using *pvsanal* to generate *fsrc* and *fdest*. These must have the same format.



### Warning

It is unsafe to use the same f-variable for both input and output of pvs opcodes. Using the same one might lead to undefined behavior on some opcodes. Use a different one on the left and right sides of the opcode.

## Examples

Here is an example of the use of the *pvscross* opcode. It uses the file *pvscross.csd* [examples/pvscross.csd].

### Example 750. Example of the *pvscross* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 16
nchnls = 1
0dbfs = 1

; example written by joachim heintz 2009

instr 1
ipermut = p4; 1 = change order of soundfiles
ifftsize = 1024
```

```
ioverlap = ifftsize / 4
iwinshape = ifftsize
iwinshape = 1; von-Hann window
Sfile1 = "fox.wav"
Sfile2 = "wave.wav"
ain1 soundin Sfile1
ain2 soundin Sfile2
fftin1 pvsanal ain1, ifftsize, ioverlap, iwinshape; fft-analysis of file 1
fftin2 pvsanal ain2, ifftsize, ioverlap, iwinshape; fft-analysis of file 2
ktrans linseg 0, p3, 1; linear transition
if ipermut == 1 then
fcross pvscross fftin2, fftin1, ktrans, 1-ktrans
else
fcross pvscross fftin1, fftin2, ktrans, 1-ktrans
endif
aout pvsynth fcross
out aout
endin

</CsInstruments>
<CsScore>
i 1 0 2.757 0; frequencies from fox.wav; amplitudes moving from wave to fox
i 1 3 2.757 1; frequencies from wave.wav, amplitudes moving from fox to wave
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvsanal*, *pvsynth*, *pvsadsyn*

## Credits

Author: Richard Dobson  
August 2001

November 2003. Thanks to Kanata Motohashi, fixed the link to the *pvcross* opcode.

New in version 4.13

# pvsdemix

pvsdemix — Spectral azimuth-based de-mixing of stereo sources.

## Description

Spectral azimuth-based de-mixing of stereo sources, with a reverse-panning result. This opcode implements the Azimuth Discrimination and Resynthesis (ADReSS) algorithm, developed by Dan Barry (Barry et Al. "Sound Source Separation Azimuth Discrimination and Resynthesis". DAFx'04, Univ. of Napoli). The source separation, or de-mixing, is controlled by two parameters: an azimuth position (*kpos*) and a subspace width (*kwidth*). The first one is used to locate the spectral peaks of individual sources on a stereo mix, whereas the second widens the 'search space', including/excluding the peaks around *kpos*. These two parameters can be used interactively to extract source sounds from a stereo mix. The algorithm is particularly successful with studio recordings where individual instruments occupy individual panning positions; it is, in fact, a reverse-panning algorithm.



### Warning

It is unsafe to use the same f-variable for both input and output of pvs opcodes. Using the same one might lead to undefined behavior on some opcodes. Use a different one on the left and right sides of the opcode.

## Syntax

```
fsig pvsdemix fleft, fright, kpos, kwidth, ipoints
```

## Performance

*fsig* -- output pv stream

*fleft* -- left channel input pv stream.

*fright* -- right channel pv stream.

*kpos* -- the azimuth target centre position, which will be de-mixed, from left to right ( $-1 \leq kpos \leq 1$ ). This is the reverse pan-pot control.

*kwidth* -- the azimuth subspace width, which will determine the number of points around *kpos* which will be used in the de-mixing process. ( $1 \leq kwidth \leq ipoints$ )

*ipoints* -- total number of discrete points, which will divide each pan side of the stereo image. This ultimately affects the resolution of the process.

## Examples

The example below takes a stereo input and passes through a de-mixing process revealing a source located at *ipos* +/- *iwidth* points. These parameters can be controlled in realtime (e.g. using FLTK widgets or MIDI) for an interactive search of sound sources.

### Example 751. Example

```
ifftsize = 1024
iwtype   = 1      /* cleaner with hanning window */
ipos     = -0.8  /* to the left of the stereo image */
iwidth   = 20    /* use peaks of 20 points around it */

al,ar    soundin  "sinput.wav"

flc      pvsanal  al, ifftsize, ifftsize/4, ifftsize, iwtype
frc      pvsanal  ar, ifftsize, ifftsize/4, ifftsize, iwtype
fdm      pvsdemix flc, frc, kpos, kwidth, 100
adm      pvsynth  fdm

        outs     adm, adm
```

## Credits

Author: Victor Lazzarini  
January 2005

New plugin in version 5

January 2005.

# pvsdiskin

pvsdiskin — Read a selected channel from a PVOC-EX analysis file.

## Description

Create an fsig stream by reading a selected channel from a PVOC-EX analysis file, with frame interpolation.

## Syntax

```
fsig pvsdiskin SFname,ktscal,kgain[,ioffset, ichan]
```

## Initialization

*Sfname* -- Name of the analysis file. This must have the .pvx file extension.

A multi-channel PVOC-EX file can be generated using the extended *pvanal utility*.

*ichan* -- (optional) The channel to read (counting from 1). Default is 1.

*ioff* -- start offset from beginning of file (secs) (default: 0) .

## Performance

*ktscal* -- time scale, ie. the read pointer speed (1 is normal speed, negative is backwards,  $0 < ktscal < 1$  is slower and  $ktscal > 1$  is faster)

*kgain* -- gain scaling.

## Examples

Here is an example of the pvsdiskin opcode. It uses the file *pvsdiskin.csd* [examples/pvsdiskin.csd].

### Example 752. Example of the pvsdiskin opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pvsdiskin.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 2
```

```
instr 1
; create a PVOC-EX (*.pvx) file with PVANAL first
ktscale line 1, p3, .05 ;change speed
fsigr pvdiskin "fox.pvx", ktscale, 1 ;read PVOCEX file
aout pvsynth fsigr ;resynthesise it
outs aout, aout

endin
</CsInstruments>
<CsScore>

i 1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
May 2007  
New in Csound 5.06

# pvsdisp

pvsdisp — Displays a PVS signal as an amplitude vs. freq graph.

## Description

This opcode will display a PVS signal fsig. Uses X11 or FLTK windows if enabled, else (or if -g flag is set) displays are approximated in ASCII characters.

## Syntax

```
pvsdisp fsig[, ibins, iwtflg]
```

## Initialization

*iprd* -- the period of pvsdisp in seconds.

*ibins* (optional, default=all bins) -- optionally, display only ibins bins.

*iwtflg* (optional, default=0) -- wait flag. If non-zero, each pvsdisp is held until released by the user. The default value is 0 (no wait).

## Performance

*pvsdisp* -- displays the PVS signal frame-by-frame.

## Examples

Here is an example of the pvsdisp opcode. It uses the file *pvsdisp.csd* [examples/pvsdisp.csd]. This example uses realtime input, but you can also use it for soundfile input.

### Example 753. Example of the pvsdisp opcode

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o pvsdisp.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 1
asig inch 1
```

```
;a1 soundin "input.wav" ;select a soundifle
fsig pvsanal asig, 1024,256, 1024, 1
pvsdisp fsig

endin

</CsInstruments>
<CsScore>

i 1 0 30
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvsanal, pvsynth, dispfft, print, pvsadsyn*

## Credits

Author: Victor Lazzarini, 2006

# pvsfilter

`pvsfilter` — Multiply amplitudes of a pvoc stream by those of a second pvoc stream, with dynamic scaling.

## Description

Multiply amplitudes of a pvoc stream by those of a second pvoc stream, with dynamic scaling.

## Syntax

```
fsig pvsfilter fsigin, fsigfil, kdepth[, igain]
```

## Performance

*fsig* -- output pv stream

*fsigin* -- input pv stream.

*fsigfil* -- filtering pvoc stream.

*kdepth* -- controls the depth of filtering of *fsigin* by *fsigfil* .

*igain* -- amplitude scaling (optional, defaults to 1).

Here the input pvoc stream amplitudes are modified by the filtering stream, keeping its frequencies intact. As usual, both signals have to be in the same format.



### Warning

It is unsafe to use the same f-variable for both input and output of pvs opcodes. Using the same one might lead to undefined behavior on some opcodes. Use a different one on the left and right sides of the opcode.

## Examples

### Example 754. Example

```
kfreq expon 500, p3, 4000           ; 3-octave sweep
kdepth linseg 1, p3/2, 0.5, p3/2, 1 ; varying filter depth

asig in                               ; input
afil oscili 1, kfreq, 1              ; filter t-domain signal

fim pvsanal asig,1024,256,1024,0     ; pvoc analysis
fil pvsanal afil,1024,256,1024,0
fou pvsfilter fim, fil, kdepth       ; filter signal
aout pvsynth fou                     ; pvoc synthesis
```

In the example above the filter curve will depend on the spectral envelope of *afil*; in the simple case of a sinusoid, it will be equivalent to a narrowband band-pass filter.

Here is an example of the use of the *pvsfilter* opcode. It uses the file *pvsfilter.csd* [examples/pvsfilter.csd].

### Example 755. Example of the *pvsfilter* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 16
nchnls = 1
0dbfs = 1

;; example written by joachim heintz 2009

giSine ftgen 0, 0, 4096, 10, 1
giBell ftgen 0, 0, 4096, 9, .56, 1, 0, .57, .67, 0, .92, 1.8, 0, .93, 1.8, 0, 1.19, 2.67, 0, 1.7, 1.6

instr 1
ipermut = p4; 1 = change order of soundfiles
ifftsize = 1024
ioverlap = ifftsize / 4
iwinshape = ifftsize
iwinshape = 1; von-Hann window
Sfile1 = "fox.wav"
ain1 soundin Sfile1
kfreq randomi 200, 300, 3
ain2 oscili .2, kfreq, giBell
;ain2 oscili .2, kfreq, giSine; try also this
fftin1 pvsanal ain1, ifftsize, ioverlap, iwinshape, iwinshape; fft-analysis of file 1
fftin2 pvsanal ain2, ifftsize, ioverlap, iwinshape, iwinshape; fft-analysis of file 2
if ipermut == 1 then
fcross pvsfilter fftin2, fftin1, 1
else
fcross pvsfilter fftin1, fftin2, 1
endif
aout pvsynth fcross
out aout * 20
endin

</CsInstruments>
<CsScore>
i 1 0 2.757 0; frequencies from fox.wav, amplitudes multiplied by amplitudes of giBell
i 1 3 2.757 1; frequencies from giBell, amplitudes multiplied by amplitudes of fox.wav
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvsanal*, *pvsynth*, *pvsadsyn*

## Credits

Author: Victor Lazzarini  
November 2004

New plugin in version 5

November 2004.

# pvsfread

pvsfread — Read a selected channel from a PVOC-EX analysis file.

## Description

Create an fsig stream by reading a selected channel from a PVOC-EX analysis file loaded into memory, with frame interpolation. Only format 0 files (amplitude+frequency) are currently supported. The operation of this opcode mirrors that of pvoc, but outputs an fsig instead of a resynthesized signal.

## Syntax

```
fsig pvsfread ktimpt, ifn [, ichan]
```

## Initialization

*ifn* -- Name of the analysis file. This must have the .pvx file extension.

A multi-channel PVOC-EX file can be generated using the extended *pvanal utility*.

*ichan* -- (optional) The channel to read (counting from 0). Default is 0.

## Performance

*ktimpt* -- Time pointer into analysis file, in seconds. See the description of the same parameter of *pvoc* for usage.

Note that analysis files can be very large, especially if multi-channel. Reading such files into memory will very likely incur breaks in the audio during real-time performance. As the file is read only once, and is then available to all other interested opcodes, it can be expedient to arrange for a dedicated instrument to preload all such analysis files at startup.

## Examples

Here is an example of the pvsfread opcode. It uses the file *pvsfread.csd* [examples/pvsfread.csd].

### Example 756. Example of the pvsfread opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pvsfread.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
```

```
ksmps = 32
0dbfs = 1
nchnls = 2

instr 1
; create a PVOC-EX (*.pvx) file with PVANAL first
idur filelen "kickroll.pvx" ;find duration of (stereo) analysis file
kpos line 0,p3,idur ;to ensure we process whole file
fsigr pvsfread kpos,"kickroll.pvx", 1 ;create fsig from right channel
aout pvsynth fsigr ;resynthesise it
outs aout, aout

endin
</CsInstruments>
<CsScore>

i 1 0 10
i 1 11 1
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Richard Dobson  
August 2001

New in version 4.13

# pvsfreeze

pvsfreeze — Freeze the amplitude and frequency time functions of a pv stream according to a control-rate trigger.

## Description

This opcode 'freezes' the evolution of pvs stream by locking into steady amplitude and/or frequency values for each bin. The freezing is controlled, independently for amplitudes and frequencies, by a control-rate trigger, which switches the freezing 'on' if equal to or above 1 and 'off' if below 1.

## Syntax

```
fsig pvsfreeze fsigin, kfreeza, kfreezf
```

## Performance

*fsig* -- output pv stream

*fsigin* -- input pv stream.

*kfreeza* -- freezing switch for amplitudes. Freezing is on if above or equal to 1 and off if below 1.

*kfreezf* -- freezing switch for frequencies. Freezing is on if above or equal to 1 and off if below 1.



### Warning

It is unsafe to use the same f-variable for both input and output of pvs opcodes. Using the same one might lead to undefined behavior on some opcodes. Use a different one on the left and right sides of the opcode.

## Examples

### Example 757. Example

```
asig in ; input
ktrig oscil 1.5, 0.25, 1 ; trigger
fim pvsanal asigl, 1024, 256, 1024, 0 ; pvoc analysis
fou pvsfreeze fim, abs(ktrig), abs(ktrig) ; regular 'freeze' of spectra
aout pvsynth fou ; pvoc synthesis
```

In the example above the input signal will be regularly 'frozen' for a short while, as the trigger rises above 1 about every two seconds.

Here is an example of the use of the *pvsfreeze* opcode. It uses the file *pvsfreeze.csd* [examples/pvsfreeze.csd].

### Example 758. Example of the *pvsfreeze* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 16
nchnls = 1
0dbfs = 1

; example written by joachim heintz 2009

seed 0

instr 1
ifftsize = 1024
ioverlap = ifftsize / 4
iwinsize = ifftsize
iwinshape = 1; von-Hann window
Sfile1 = "fox.wav"
ain soundin Sfile1
kfreq randomh .7, 1.1, 3; probability of freezing freqs: 1/4
kamp randomh .7, 1.1, 3; idem for amplitudes
fftin pvsanal ain, ifftsize, ioverlap, iwinsize, iwinshape; fft-analysis of file
freeze pvsfreeze fftin, kamp, kfreq; freeze amps or freqs independently
aout pvsynth freeze; resynthesize
out aout
endin

</CsInstruments>
<CsScore>
r 10
i 1 0 2.757
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvsanal, pvsynth, pvsadsyn*

## Credits

Author: Victor Lazzarini  
May 2006

New plugin in version 5

May 2006.

# pvsftr

pvsftr — Reads amplitude and/or frequency data from function tables.

## Description

Reads amplitude and/or frequency data from function tables.

## Syntax

```
pvsftr fsrc, ifna [, ifnf]
```

## Initialization

*ifna* -- A table, at least inbins in size, that stores amplitude data. Ignored if *ifna* = 0

*ifnf* (optional) -- A table, at least inbins in size, that stores frequency data. Ignored if *ifnf* = 0

## Performance

*fsrc* -- a PVOC-EX formatted source.

Enables the contents of *fsrc* to be exchanged with function tables for custom processing. Except when the frame overlap equals *ksmps* (which will generally not be the case), the frame data is not updated each control period. The data in *ifna*, *ifnf* should only be processed when *kflag* is set to 1. To process only frequency data, set *ifna* to zero.

As the function tables are required only to store data from *fsrc*, there is no advantage in defining them in the score, and they should generally be created in the instrument, using *ftgen*.

By exporting amplitude data, say, from one fsig and importing it into another, basic cross-synthesis (as in *pvsfcross*) can be performed, with the option to modify the data beforehand using the table manipulation opodes.

Note that the format data in the source fsig is not written to the tables. This therefore offers a means of transferring amplitude and frequency data between non-identical fsigs. Used this way, these opcodes become potentially pathological, and can be relied upon to produce unexpected results. In such cases, resynthesis using *pvsadsyn* would almost certainly be required.

To perform a straight copy from one fsig to another one of identical format, the conventional assignment syntax can be used:

```
fsig1 = fsig2
```

It is not necessary to use function tables in this case.

## Examples

Here is an example of the pvsftr opcode. It uses the file *pvsftr.csd* [examples/pvsftr.csd].

## Example 759. Example of the pvsftr opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o pvsftr.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gifil ftgen 0, 0, 0, -1, "fox.wav", 0, 0, 1

instr 1

ifftsize = 1024   ;fft size
ioverlap = 256   ;overlap
knewamp = 0      ;new value for amplitudes

;create fsig stream from function table
fsrc pvstanal 1, 1, 1, gifil, 0, 0, 0, ifftsize, ioverlap, 0
ifn ftgen 0, 0, ifftsize/2, 2, 0 ;create empty function table
kflag pvsftw fsrc,ifn ;export amps to table
;overwrite the first 10 bins each time the table has been filled new
if kflag == 1 then
kndx = 0
kmaxbin = 10
loop:
tablew knewamp, kndx, ifn
loop_le kndx, 1, kmaxbin, loop
endif
pvsftr fsrc,ifn ;read modified data back to fsrc
aout pvsynth fsrc ;and resynth
outs aout, aout

endin
</CsInstruments>
<CsScore>
i 1 0 4
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvsftw*

## Credits

Author: Richard Dobson  
August 2001

New in version 4.13

# pvsftw

pvsftw — Writes amplitude and/or frequency data to function tables.

## Description

Writes amplitude and/or frequency data to function tables.

## Syntax

```
kflag pvsftw fsrc, ifna [, ifnf]
```

## Initialization

*ifna* -- A table, at least inbins in size, that stores amplitude data. Ignored if *ifna* = 0

*ifnf* -- A table, at least inbins in size, that stores frequency data. Ignored if *ifnf* = 0

## Performance

*kflag* -- A flag that has the value of 1 when new data is available, 0 otherwise.

*fsrc* -- a PVOC-EX formatted source.

Enables the contents of *fsrc* to be exchanged with function tables, for custom processing. Except when the frame overlap equals *ksmps* (which will generally not be the case), the frame data is not updated each control period. The data in *ifna*, *ifnf* should only be processed when *kflag* is set to 1. To process only frequency data, set *ifna* to zero.

As the functions tables are required only to store data from *fsrc*, there is no advantage in defining them in the score. They should generally be created in the instrument using *ftgen*.

By exporting amplitude data, say, from one fsig and importing it into another, basic cross-synthesis (as in *pvcross*) can be performed, with the option to modify the data beforehand using the table manipulation opodes.

Note that the format data in the source fsig is not written to the tables. This therefore offers a means of transferring amplitude and frequency data between non-identical fsigs. Used this way, these opcodes become potentially pathological, and can be relied upon to produce unexpected results. In such cases, resynthesis using *pvsadsyn* would almost certainly be required.

To perform a straight copy from one fsig to another one of identical format, the conventional assignment syntax can be used:

```
fsig1 = fsig2
```

It is not necessary to use function tables in this case.

## Examples

Here is an example of the pvsftw opcode. It uses the file *pvsftw.csd* [examples/pvsftw.csd].

## Example 760. Example of the pvsftw opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pvsftw.wav -W   ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

inbins = 512
ifn ftgen 0,0,inbins,10,1 ; make ftable
fsrc pvsdiskin "fox.pvx", 1, 1 ; read PVOCEX file
kflag pvsftw fsrc,ifn ; export amps to table,
kamp init 0
if kflag==0 kgoto contin ; only proc when frame is ready
tablew kamp,1,ifn ; kill lowest bins, for obvious effect
tablew kamp,2,ifn
tablew kamp,3,ifn
tablew kamp,4,ifn
; read modified data back to fsrc
pvsftr fsrc,ifn
contin:
; and resynth
aout pvsynth fsrc
outs aout, aout

endin

</CsInstruments>
<CsScore>

i 1 0 4
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvsftr*

## Credits

Author: Richard Dobson  
August 2001

New in version 4.13

# pvsfwrite

pvsfwrite — Write a fsig to a PVOCEX file.

## Description

This opcode writes a fsig to a PVOCEX file (which in turn can be read by pvsfread or other programs that support PVOCEX file input).

## Syntax

```
pvsfwrite fsig, ifile
```

## Initialisation

*fsig* -- fsig input data. *ifile* -- filename (a string in double-quotes) .

## Examples

Here is an example of the pvsfwrite opcode. It uses the file *pvsfwrite.csd* [examples/pvsfwrite.csd]. This example uses realtime audio input.

### Example 761. Example of the pvsfwrite opcode

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o pvsfwrite.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

;By Victor Lazzarini 2008

instr 1
asig oscili 10000, 440, 1
fss pvsanal asig, 1024,256,1024,0
pvsfwrite fss, "mypvs.pvx"
ase pvsynth fss
      out ase
endin

instr 2 ; must be called after instr 1 finishes
ktim timeinsts
fss pvsfread ktim, "mypvs.pvx"
asig pvsynth fss
```

```
    out asig
  endin

</CsInstruments>
<CsScore>
f1 0 16384 10 1
i1 0 1
i2 1 1
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvsanal*, *pvsynth*, *pvsadsyn*

## Credits

Author: Victor Lazzarini  
November 2004

New plugin in version 5

November 2004.

# pvsgain

pvsgain — Scale the amplitude of a pv stream.

## Description

Scale the amplitude of a pv stream.

## Syntax

```
fsig pvsgain fsigin, kgain
```

## Performance

*fsig* -- output pv stream

*fsigin* -- input pv stream

*kgain* -- amplitude scaling (defaults to 1).



### Warning

It is unsafe to use the same f-variable for both input and output of pvs opcodes. Using the same one might lead to undefined behavior on some opcodes. Use a different one on the left and right sides of the opcode.

## Examples

Here is an example of the use of the *pvsgain* opcode. It uses the file *pvsgain.csd* [examples/pvsgain.csd].

### Example 762. Example of the *pvsgain* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pvsgain.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kgain = p4
asig diskin2 "beats.wav", 1
```

```
fsig pvsanal asig, 1024, 256, 1024, 1; analyse it
ftps pvsgain fsig, kgain ; amplify it
atps pvsynth ftps ; synthesise it
outs      atps, atps

endin
</CsInstruments>
<CsScore>

i1 0 2 .5
i1 + 2 1

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvsanal, pvsynth, pvsadsyn*

## Credits

Author: Victor Lazzarini  
2011

New plugin in version 5

2011

# pvshift

pvshift — Shift the frequency components of a pv stream, stretching/compressing its spectrum.

## Description

Shift the frequency components of a pv stream, stretching/compressing its spectrum.

## Syntax

```
fsig pvshift fsigin, kshift, klowest[, kkeepform, igain, kcoefs]
```

## Performance

*f`sig`* -- output pv stream

*f`sigin`* -- input pv stream

*k`shift`* -- shift amount (in Hz, positive or negative).

*k`lowest`* -- lowest frequency to be shifted.

*k`keepform`* -- attempt to keep input signal formants; 0: do not keep formants; 1: keep formants using a liftered cepstrum method; 2: keep formants by using a true envelope method (defaults to 0).

*k`gain`* -- amplitude scaling (defaults to 1).

*k`coefs`* -- number of cepstrum coefs used in formant preservation (defaults to 80).

This opcode will shift the components of a pv stream, from a certain frequency upwards, up or down a fixed amount (in Hz). It can be used to transform a harmonic spectrum into an inharmonic one. The *k`keepform`* flag can be used to try and preserve formants for possibly interesting and unusual spectral modifications.



### Warning

It is unsafe to use the same f-variable for both input and output of pvs opcodes. Using the same one might lead to undefined behavior on some opcodes. Use a different one on the left and right sides of the opcode.

## Examples

### Example 763. Example

```
asig in ; get the signal in

fsig pvsanal asig, 1024, 256, 1024, 1 ; analyse it
ftps pvshift fsig, 100, 0 ; add 100 Hz to each component
atps pvsynth ftps ; synthesise it
```

Depending on the input, this will transform a pitched sound into an inharmonic, bell-like sound.

Here is an example of the use of the *pvshift* opcode. It uses the file *pvshift.csd* [examples/pvshift.csd].

## Example 764. Example of the *pvshift* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 16
nchnls = 1
0dbfs = 1

; example written by joachim heintz 2009

instr 1
ishift = p4; shift amount in Hz
ilowest = p5; lowest frequency to be shifted
ikeepform = p6; 0=no formant keeping, 1=keep by amps, 2=keep by spectral envelope
ifftsize = 1024
ioverlap = ifftsize / 4
iwinshape = ifftsize
iwinshape = 1; von-Hann window
Sfile = "fox.wav"
ain soundin Sfile
fftin pvsanal ain, ifftsize, ioverlap, iwinshape; fft-analysis of file
fshift pvshift fftin, ishift, ilowest, ikeepform; shift frequencies
aout pvsynth fshift; resynthesize
out aout
endin

</CsInstruments>
<CsScore>
i 1 0 2.757 0 0 0; no shift at all
i 1 3 2.757 100 0 0; shift all frequencies by 100 Hz
i 1 6 2.757 200 0 0; by 200 Hz
i 1 9 2.757 200 0 1; keep formants by method 1
i 1 12 2.757 200 0 2; by method 2
i 1 15 2.757 200 1000 0; shift by 200 Hz but just above 1000 Hz
i 1 18 2.757 1000 500 0; shift by 1000 Hz above 500 Hz
i 1 21 2.757 1000 300 0; above 300 Hz
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvsanal*, *pvsynth*, *pvsadsyn*

## Credits

Author: Victor Lazzarini  
November 2004

New plugin in version 5

November 2004.

# pvsifd

pvsifd — Instantaneous Frequency Distribution, magnitude and phase analysis.

## Description

The pvsifd opcode takes an input a-rate signal and performs an Instantaneous Frequency, magnitude and phase analysis, using the STFT and pvsifd (Instantaneous Frequency Distribution), as described in Lazzarini et al, "Time-stretching using the Instantaneous Frequency Distribution and Partial Tracking", Proc.of ICMC05, Barcelona. It generates two PV streaming signals, one containing the amplitudes and frequencies (a similar output to pvsanal) and another containing amplitudes and unwrapped phases.

## Syntax

```
ffr,fphs pvsifd ain, ifftsize, ihopsize, iwintype[,iscal]
```

## Performance

*ffr* -- output pv stream in AMP\_FREQ format

*fphs* -- output pv stream in AMP\_PHASE format

*ifftsize* -- FFT analysis size, must be power-of-two and integer multiple of the hopsize.

*ihopsize* -- hopsize in samples

*iwintype* -- window type (O: Hamming, 1: Hanning)

*iscal* -- amplitude scaling (defaults to 1).



### Warning

It is unsafe to use the same f-variable for both input and output of pvs opcodes. Using the same one might lead to undefined behavior on some opcodes. Use a different one on the left and right sides of the opcode.

## Examples

Here is an example of the pvsifd opcode. It uses the file *pvsifd.csd* [examples/pvsifd.csd].

### Example 765. Example of the pvsifd opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pvsifd.wav -W ;; for file output any platform
```

```
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ain diskin2 "beats.wav", 1, 0, 1
fs1,fsi2 pvsifd ain, 2048, 512, 1 ; pvsifd analysis
fst partials fs1, fsi2, .1, 1,3, 500 ; partial tracking
aout resyn fst, 1, 1.5, 500, 1 ; resynthesis (up a 5th)
outs aout, aout

endin
</CsInstruments>
<CsScore>
;sine
f1 0 4096 10 1

i 1 0 2
e
</CsScore>
</CsoundSynthesizer>
```

The example above shows the pvsifd analysis feeding into partial tracking and cubic-phase additive resynthesis with pitch shifting.

## Credits

Author: Victor Lazzarini  
June 2005

New plugin in version 5

November 2004.

# pvsinfo

pvsinfo — Get information from a PVOC-EX formatted source.

## Description

Get format information about fsrc, whether created by an opcode such as pvsanal, or obtained from a PVOCEX file by pvsfread. This information is available at init time, and can be used to set parameters for other pvs opcodes, and in particular for creating function tables (e.g. for pvsftw), or setting the number of oscillators for pvsadsyn.

## Syntax

```
ioverlap, inumbins, iwinsize, iformat pvsinfo fsrc
```

## Initialization

*ioverlap* -- The stream overlap size.

*inumbins* -- The number of analysis bins (amplitude+frequency) in fsrc. The underlying FFT size is calculated as  $(inumbins - 1) * 2$ .

*iwinsize* -- The analysis window size. May be larger than the FFT size.

*iformat* -- The analysis frame format. If fsrc is created by an opcode, iformat will always be 0, signifying amplitude+frequency. If fsrc is defined from a PVOC-EX file, iformat may also have the value 1 or 2 (amplitude+phase, complex).

## Examples

Here is an example of the pvsinfo opcode. It uses the file *pvsinfo.csd* [examples/pvsinfo.csd].

### Example 766. Example of the pvsinfo opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o pvsinfo.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
; create a PVOC-EX (*.pvx) file with PVANAL first
```

```
idur filelen "fox.pvx" ;find duration of analysis file
kpos line 0,p3,idur ;to ensure we process whole file
fsrc pvsfread kpos, "fox.pvx" ;create fsig from (mono) file

iovl,inb,iws,ifmt pvsinfo fsrc ;get info
print iovl,inb,iws,ifmt ;print info

aout pvsynth fsrc
outs aout, aout

endin
</CsInstruments>
<CsScore>

i 1 0 3
e
</CsScore>
</CsoundSynthesizer>
```

The example will produce the following output:

```
instr 1: iovl = 256.000 inb = 513.000 iws = 2048.000 ifmt = 0.000
```

## Credits

Author: Richard Dobson  
August 2001

New in version 4.13

# pvsinit

pvsinit — Initialise a spectral (f) variable to zero.

## Description

Performs the equivalent to an init operation on an f-variable.

## Syntax

```
fsig pvsinit isize[, iolap, iwinsize, iwintype, iformat]
```

## Performance

*fsig* -- output pv stream set to zero.

*isize* -- size of the DFT frame.

*iolap* -- size of the analysis overlap, defaults to *isize*/4.

*iwinsize* -- size of the analysis window, defaults to *isize*.

*iwintype* -- type of analysis window, defaults to 1, Hanning.

*iformat* -- pvsdata format, defaults to 0:PVS\_AMP\_FREQ.

## Examples

### Example 767. Example

```
fsig pvsinit 1024
```

## Credits

Author: Victor Lazzarini  
November 2004

New plugin in version 5

November 2004.

# pvsin

*pvsin* — Retrieve an f-sig from the input software bus; a pvs equivalent to *chani*.

## Description

This opcode retrieves an f-sig from the pvs in software bus, which can be used to get data from an external source, using the Csound 5 API. A channel is created if not already existing. The f-sig channel is in that case initialised with the given parameters. It is important to note that the pvs input and output (*pvsout* opcode) busses are independent and data is not shared between them.

## Syntax

```
f-sig pvsin kchan[, isize, iolap, iwinsize, iwintype, iformat]
```

## Initialization

*isize* -- initial DFT size, defaults to 1024.

*iolap* -- size of overlap, defaults to *isize*/4.

*iwinsize* -- size of analysis window, defaults to *isize*.

*iwintype* -- type of window, defaults to Hanning (1) (see *pvsanal*)

*iformat* -- data format, defaults 0 (PVS\_AMP\_FREQ). Other possible values are 1 (PVS\_AMP\_PHASE), 2 (PVS\_COMPLEX) or 3 (PVS\_TRACKS).

## Performance

*f-sig* -- output f-sig.

*kchan* -- channel number. If non-existent, a channel will be created.

## Examples

### Example 768. Example

```
f-sig pvsin 0 ; get data from pvs in bus channel 0
```

## Credits

Author: Victor Lazzarini  
August 2006

# pvslock

pvslock — Frequency lock an input fsig

## Description

This opcode searches for spectral peaks and then locks the frequencies around those peaks. This is similar to phase-locking in non-streaming PV processing. It can be used to improve timestretching and pitch-shifting quality in PV processing.

## Syntax

```
fsig pvslock fsigin, klock
```

## Performance

*fsig* -- output pv stream

*fsigin* -- input pv stream.

*klock* -- frequency lock, 1 -> lock, 0 -> unlock (bypass).



### Warning

It is unsafe to use the same f-variable for both input and output of pvs opcodes. Using the same one might lead to undefined behavior on some opcodes. Use a different one on the left and right sides of the opcode.

## Examples

Here is an example of the pvslock opcode. It uses the file *pvslock.csd* [examples/pvslock.csd].

### Example 769. Example of the pvslock opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pvslock.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gifil ftgen 0, 0, 0, 1, "fox.wav", 0, 0, 1
```

```
instr 1

klock = p4
fsig   pvstanal 1, 1, 1, gfil ; no further transformations
fsigout pvslock fsig, klock ; lock frequency
aout   pvsynth fsigout
        outs    aout, aout

endin
</CsInstruments>
<CsScore>

i 1 0 2.6 1 ; locked
i 1 3 2.6 0 ; not locked

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
November 2004

New plugin in version 5

November 2004.

# pvsmaska

pvsmaska — Modify amplitudes using a function table, with dynamic scaling.

## Description

Modify amplitudes of fsrc using function table, with dynamic scaling.

## Syntax

```
fsig pvsmaska fsrc, ifn, kdepth
```

## Initialization

*ifn* -- The f-table to use. Given fsrc has N analysis bins, table ifn must be of size N or larger. The table need not be normalized, but values should lie within the range 0 to 1. It can be supplied from the score in the usual way, or from within the orchestra by using *pvsinfo* to find the size of fsrc, (returned by pvsinfo in inbins), which can then be passed to ftgen to create the f-table.

## Performance

*kdepth* -- Controls the degree of modification applied to fsrc, using simple linear scaling. 0 leaves amplitudes unchanged, 1 applies the full profile of ifn.

Note that power-of-two FFT sizes are particularly convenient when using table-based processing, as the number of analysis bins (inbins) is then a power-of-two plus one, for which an exactly matching f-table can be created. In this case it is important that the f-table be created with a size of inbins, rather than as a power of two, as the latter will copy the first table value to the guard point, which is inappropriate for this opcode.



### Warning

It is unsafe to use the same f-variable for both input and output of pvs opcodes. Using the same one might lead to undefined behavior on some opcodes. Use a different one on the left and right sides of the opcode.

## Examples

Here is an example of the use of the *pvsmaska* opcode. It uses the file *pvsmaska.csd* [examples/pvsmaska.csd].

### Example 770. Example of the *pvsmaska* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
-odac  
</CsOptions>  
<CsInstruments>
```

```
sr = 44100
ksmps = 16
nchnls = 1
0dbfs = 1

; ; example written by joachim heintz 2009

; ; function table for defining amplitude peaks (from the example of Richard Dobson)
giTab ftgen 0, 0, 513, 8, 0, 2, 1, 3, 0, 4, 1, 6, 0, 10, 1, 12, 0, 16, 1, 32, 0, 1, 0, 436, 0

instr 1
imod = p4; degree of midification (0-1)
ifftsize = 1024
ioverlap = ifftsize / 4
iwinsize = ifftsize
iwinshape = 1; von-Hann window
Sfile = "fox.wav"
ain soundin Sfile
fftin pvsanal ain, ifftsize, ioverlap, iwinsize, iwinshape; fft-analysis of file
fmask pvsmaska fftin, giTab, imod
aout pvsynth fmask; resynthesize
  out aout
endin

</CsInstruments>
<CsScore>
i 1 0 2.757 0
i 1 3 2.757 1
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvsanal, pvsynth, pvsadsyn*

## Credits

Author: Richard Dobson  
August 2001

New in version 4.13

# pvmix

pvmix — Mix 'seamlessly' two pv signals.

## Description

Mix 'seamlessly' two pv signals. This opcode combines the most prominent components of two pvoc streams into a single mixed stream.

## Syntax

```
fsig pvmix fsigin1, fsigin2
```

## Performance

*fsig* -- output pv stream

*fsigin1* -- input pv stream.

*fsigin2* -- input pv stream, which must have same format as *fsigin1*.



### Warning

It is unsafe to use the same f-variable for both input and output of pvs opcodes. Using the same one might lead to undefined behavior on some opcodes. Use a different one on the left and right sides of the opcode.

## Examples

Here is an example of the pvmix opcode. It uses the file *pvmix.csd* [examples/pvmix.csd].

### Example 771. Example of the pvmix opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pvmix.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gisin ftgen 1, 0, 2048, 10, 1

instr 1
```

```
asig1 diskin2 "fox.wav", 1 ;signal in 1
asig2 oscil .3, 100, gisin ;signal in 2
fsig1 pvsanal asig1,1024,256,1024,0 ;pvoc analysis
fsig2 pvsanal asig2,1024,256,1024,0 ;of both signals
fsall pvmix fsig1, fsig2
asig pvsynth fsall
      outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 3
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
November 2004

New plugin in version 5

November 2004.

# pvsmorph

pvsmorph — Performs morphing (or interpolation) between two source fsigs.

## Description

Performs morphing (or interpolation) between two source fsigs.

## Syntax

```
fsig pvsmorph fsig1, fsig2, kampint, kfrqint
```

## Performance

The operation of this opcode is similar to that of *pvinterp* (q.v.), except in using *fsigs* rather than analysis files, and the absence of spectral envelope preservation. The amplitudes and frequencies of *fsig1* are interpolated with those of *fsig2*, depending on the values of *kampint* and *kfrqint*, respectively. These range between 0 and 1, where 0 means *fsig1* and 1, *fsig2*. Anything in between will interpolate amps and/or freqs of the two fsigs.

With this opcode, morphing can be performed on real-time audio input, by using *pvsanal* to generate *fsig1* and *fsig2*. These must have the same format.



### Warning

It is unsafe to use the same f-variable for both input and output of pvs opcodes. Using the same one might lead to undefined behavior on some opcodes. Use a different one on the left and right sides of the opcode.

## Examples

Here is an example of the pvsmorph opcode. It uses the file *pvsmorph.csd* [examples/pvsmorph.csd].

### Example 772. Example of the pvsmorph opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o pvsmorph.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 16
nchnls = 1
0dbfs = 1
```

```

;; example written by joachim heintz 2009

giSine ftgen 0, 0, 4096, 10, 1

instr 1
iampint1 = p4
iampint2 = p5
ifrqint1 = p6
ifrqint2 = p7
kampint linseg iampint1, p3, iampint2
kfrqint linseg ifrqint1, p3, ifrqint2
ifftsize = 1024
ioverlap = ifftsize / 4
iwinsize = ifftsize
iwinshape = 1; von-Hann window
Sfile1 = "fox.wav"
ain1 soundin Sfile1
ain2 buzz .2, 50, 100, giSine
fftin1 pvsanal ain1, ifftsize, ioverlap, iwinsize, iwinshape
fftin2 pvsanal ain2, ifftsize, ioverlap, iwinsize, iwinshape
fmorph pvsmorph fftin1, fftin2, kampint, kfrqint
aout pvsynth fmorph
out aout * .5
endin

</CsInstruments>
<CsScore>
i 1 0 3 0 0 1 1
i 1 3 3 1 0 1 0
i 1 6 3 0 1 0 1
e
</CsScore>
</CsoundSynthesizer>

```

Here is another example of the pvsmorph opcode. It uses the file *pvsmorph2.csd* [examples/pvsmorph2.csd].

### Example 773. Example of the pvsmorph opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 16
nchnls = 1
0dbfs = 1

;; example written by joachim heintz 2009
;; this example uses the files "flute-C-octave0.wav" and
;; "saxophone-alto-C-octave0.wav" from www.archive.org/details/OpenPathMusic44V2

giSine ftgen 0, 0, 4096, 10, 1

instr 1
iampint1 = p4; value for interpolating the amplitudes at the beginning ...
iampint2 = p5; ... and at the end
ifrqint1 = p6; value for interpolating the frequencies at the beginning ...
ifrqint2 = p7; ... and at the end
kampint linseg iampint1, p3, iampint2

```

```

kfrqint linseg ifrqint1, p3, ifrqint2
ifftsize = 1024
ioverlap = ifftsize / 4
iwinsize = ifftsize
iwinshape = 1; von-Hann window
Sfile1 = "flute-C-octave0.wav"
Sfile2 = "saxophone-alto-C-octave0.wav"
ain1 soundin Sfile1
ain2 soundin Sfile2
fftin1 pvsanal ain1, ifftsize, ioverlap, iwinsize, iwinshape
fftin2 pvsanal ain2, ifftsize, ioverlap, iwinsize, iwinshape
fmorph pvsmorph fftin1, fftin2, kampint, kfrqint
aout pvsynth fmorph
    out aout * .5
endin

instr 2; moving randomly in certain borders between two spectra
iampintmin = p4; minimum value for amplitudes
iampintmax = p5; maximum value for amplitudes
ifrqintmin = p6; minimum value for frequencies
ifrqintmax = p7; maximum value for frequencies
imovefreq = p8; frequency for generating new random values
kampint randomi iampintmin, iampintmax, imovefreq
kfrqint randomi ifrqintmin, ifrqintmax, imovefreq
ifftsize = 1024
ioverlap = ifftsize / 4
iwinsize = ifftsize
iwinshape = 1; von-Hann window
Sfile1 = "flute-C-octave0.wav"
Sfile2 = "saxophone-alto-C-octave0.wav"
ain1 soundin Sfile1
ain2 soundin Sfile2
fftin1 pvsanal ain1, ifftsize, ioverlap, iwinsize, iwinshape
fftin2 pvsanal ain2, ifftsize, ioverlap, iwinsize, iwinshape
fmorph pvsmorph fftin1, fftin2, kampint, kfrqint
aout pvsynth fmorph
    out aout * .5
endin

</CsInstruments>
<CsScore>
i 1 0 3 0 0 1 1; amplitudes from flute, frequencies from saxophone
i 1 3 3 1 1 0 0; amplitudes from saxophone, frequencies from flute
i 1 6 3 0 1 0 1; amplitudes and frequencies moving from flute to saxophone
i 1 9 3 1 0 1 0; amplitudes and frequencies moving from saxophone to flute
i 2 12 3 .2 .8 .2 .8 5; amps and freqs moving randomly between the two spectra
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*pvsanal*, *pvsynth*, *pvsadsyn*

## Credits

Author: Victor Lazzarini  
 April 2007  
 New in Csound 5.06

# pvsmooth

pvsmooth — Smooth the amplitude and frequency time functions of a pv stream using parallel 1st order lowpass IIR filters with time-varying cutoff frequency.

## Description

Smooth the amplitude and frequency time functions of a pv stream using a 1st order lowpass IIR with time-varying cutoff frequency. This opcode uses the same filter as the *tone* opcode, but this time acting separately on the amplitude and frequency time functions that make up a pv stream. The cutoff frequency parameter runs at the control-rate, but unlike *tone* and *tonek*, it is not specified in Hz, but as fractions of 1/2 frame-rate (actually the pv stream sampling rate), which is easier to understand. This means that the highest cutoff frequency is 1 and the lowest 0; the lower the frequency the smoother the functions and more pronounced the effect will be.

These are filters applied to control signals so the effect is basically blurring the spectral evolution. The effects produced are more or less similar to *pvsblur*, but with two important differences: 1. smoothing of amplitudes and frequencies use separate sets of filters; and 2. there is no increase in computational cost when higher amounts of 'blurring' (smoothing) are desired.

## Syntax

```
fsig pvsmooth fsigin, kacf, kkcf
```

## Performance

*f<sub>sig</sub>* -- output pv stream

*f<sub>sigin</sub>* -- input pv stream.

*k<sub>acf</sub>* -- amount of cutoff frequency for amplitude function filtering, between 0 and 1, in fractions of 1/2 frame-rate.

*k<sub>kcf</sub>* -- amount of cutoff frequency for frequency function filtering, between 0 and 1, in fractions of 1/2 frame-rate.



### Warning

It is unsafe to use the same f-variable for both input and output of pvs opcodes. Using the same one might lead to undefined behavior on some opcodes. Use a different one on the left and right sides of the opcode.

## Examples

Here is an example of the pvsmooth opcode. It uses the file *pvsmooth.csd* [examples/pvsmooth.csd].

### Example 774. Example of the pvsmooth opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac ;;realtime audio out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pvsmooth.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kacf = p4
kfcf = p5
asig soundin "fox.wav"
fsig pvsanal asig, 1024, 256, 1024, 1 ; analyse it
ftps pvsmooth fsig, kacf, kfcf
atps pvsynth ftps ; synthesise it
outs atps*3, atps*3

endin
</CsInstruments>
<CsScore>
; amp freq
i 1 0 3 0.01 0.01 ;smooth amplitude and frequency with cutoff frequency of filter at 1% of 1/2 frame-ra
i 1 + 3 1 0.01 ;no smoothing on amplitude, but frequency with cf at 1% of 1/2 frame-rate (ca 0.86 Hz)
i 1 + 10 .001 1 ;smooth amplitude with cf at 0.1% of 1/2 frame-rate (ca 0.086 Hz)
;and no smoothing of frequency
e
</CsScore>
</CsoundSynthesizer>
```

The formula for calculating the cutoff frequency of the filter: frame rate / hopsize = new frame starts per second (in Hz), then the percentage of half the framerate. For example, looking at the first note in the example, the frame rate is  $44100 / 256 = 172,265625$  Hz (= 172 new frame starts per second). half of the frame rate is about 86 Hz, and one percent of this is 0.86 Hz.

## Credits

Author: Victor Lazzarini  
May 2006

New plugin in version 5

May 2006.

# pvsout

pvsout — Write a fsig to the pvs output bus.

## Description

This opcode writes a fsig to a channel of the pvs output bus. Note that the pvs out bus and the pvs in bus are separate and independent. A new channel is created if non-existent.

## Syntax

```
pvsout fsig, kchan
```

## Performance

*fsig* -- fsig input data.

*kchan* -- pvs out bus channel number.

## Examples

### Example 775. Example

```
asig      in                               ; input
fsig      pvsanal asig, 1024, 256, 1024, 1 ; analysis
          pvsout fsig, 0                   ; write signal to pvs out bus channel 0
```

## Credits

Author: Victor Lazzarini  
August 2006

# pvsosc

pvsosc — PVS-based oscillator simulator.

## Description

Generates periodic signal spectra in AMP-FREQ format, with the option of four wave types:

1. sawtooth-like (harmonic weight  $1/n$ , where  $n$  is partial number)
2. square-like (similar to 1., but only odd partials)
3. pulse (all harmonics with same weight)
4. cosine

Complex waveforms (ie. all types except cosine) contain all harmonics up to the Nyquist. This makes pvsosc an option for generation of band-limited periodic waves. In addition, types can be changed using a k-rate variable.

## Syntax

```
fsig pvsosc kamp, kfreq, ktype, isize [, ioverlap] [, iwinsize] [, iwintype] [, iformat]
```

## Initialisation

*fsig* -- output pv stream set to zero.

*isize* -- size of analysis frame and window.

*ioverlap* -- (Optional) size of overlap, defaults to  $isize/4$ .

*iwinsize* -- (Optional) window size, defaults to  $isize$ .

*iwintype* -- (Optional) window type, defaults to Hanning. The choices are currently:

- 0 = Hamming window
- 1 = von Hann window

*iformat* -- (Optional) data format, defaults to 0 which produces AMP:FREQ data. That is currently the only option.

## Performance

*kamp* -- signal amplitude. Note that the actual signal amplitude can, depending on wave type and frequency, vary slightly above or below this value. Generally the amplitude will tend to exceed *kamp* on higher frequencies (> 1000 Hz) and be reduced on lower ones. Also due to the overlap-add process, when resynthesising with pvsynth, frequency glides will cause the output amplitude to fluctuate above and below *kamp*.

*kfreq* -- fundamental frequency in Hz.

*ktype* -- wave type: 1. sawtooth-like, 2. square-like, 3. pulse and any other value for cosine.

## Examples

Here is an example of the `pvsosc` opcode. It uses the file `pvsosc.csd` [examples/pvsosc.csd].

### Example 776. Example of the `pvsosc` opcode

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o pvsosc.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 1
; a band-limited sawtooth-wave oscillator
fsig pvsosc 10000, 440, 1, 1024 ; generate wave spectral signal
asig pvsynth fsig                ; resynthesize it
out asig
endin

instr 2
; a band-limited square-wave oscillator
fsig pvsosc 10000, 440, 2, 1024 ; generate wave spectral signal
asig pvsynth fsig                ; resynthesize it
out asig
endin

instr 3
; a pulse oscillator
fsig pvsosc 10000, 440, 3, 1024 ; generate wave spectral signal
asig pvsynth fsig                ; resynthesize it
out asig
endin

instr 4
; a cosine-wave oscillator
fsig pvsosc 10000, 440, 4, 1024 ; generate wave spectral signal
asig pvsynth fsig                ; resynthesize it
out asig
endin

</CsInstruments>
<CsScore>

i 1 0 1
i 2 2 1
i 3 4 1
i 4 6 1

e

</CsScore>

```

`</CsoundSynthesizer>`

## See Also

*pvsanal, pvsynth, pvsadsyn*

## Credits

Author: Victor Lazzarini  
August 2006

# pvspitch

pvspitch — Track the pitch and amplitude of a PVS signal.

## Description

Track the pitch and amplitude of a PVS signal as k-rate variables.

## Syntax

```
kfr, kamp pvspitch fsig, kthresh
```

## Performance

*kamp* -- Amplitude of fundamental frequency

*kfr* -- Fundamental frequency

*fsig* -- an input pv stream

*kthresh* -- analysis threshold (between 0 and 1). Higher values will eliminate low-amplitude components from the analysis.

## Performance

The pitch detection algorithm implemented by *pvspitch* is based upon J. F. Schouten's hypothesis of the neural processes of the brain used to determine the pitch of a sound after the frequency analysis of the basilar membrane. Except for some further considerations, *pvspitch* essentially seeks out the highest common factor of an incoming sound's spectral peaks to find the pitch that may be attributed to it.

In general, input sounds that exhibit pitch will also exhibit peaks in their spectrum according to where their harmonics lie. There are some the exceptions, however. Some sounds whose spectral representation is continuous can impart a sensation of pitch. Such sounds are explained by the centroid or center of gravity of the spectrum and are beyond the scope of the method of pitch detection implemented by *pvspitch* (Using opcodes like *pvscent* might be more appropriate in these cases).

*pvspitch* is able (using a previous analysis *fsig* generated by *pvsanal*) to locate the spectral peaks of a signal. The threshold parameter (*kthresh*) is of utmost importance, as adjusting it can introduce weak yet significant harmonics into the calculation of the fundamental. However, bringing *kthresh* too low would allow harmonically unrelated partials into the analysis algorithm and this will compromise the method's accuracy. These initial steps emulate the response of the basilar membrane by identifying physical characteristics of the input sound. The choice of *kthresh* depends on the actual level of the input signal, since its range (from 0 to 1) spans the whole dynamic range of an analysis bin (from -inf to 0dBFS).

It is important to remember that the input to the *pvspitch* opcode is assumed to be characterised by strong partials within its spectrum. If this is not the case, the results outputted by the opcode may not bear any relation to the pitch of the input signal. If a spectral frame with many unrelated partials was analysed, the greatest common factor of these frequency values that allows for adjacent “harmonics” would be chosen. Thus, noisy frames can be characterised by low frequency outputs of *pvspitch*. This fact allows for a primitive type of instrumental transient detection, as the attack portion of some instrumental tones contain inharmonic components. Should the lowest frequency of the analysed melody be known, then all frequencies detected below this threshold are inaccurate readings, due to the presence of unrelated partials.

In order to facilitate efficient testing of the *pvspitch* algorithm, an amplitude value proportional to the one in the observed in the signal frame is also outputted (*kamp*). The results of *pvspitch* can then be employed to drive an oscillator whose pitch can be audibly compared with that of the original signal (In the example below, an oscillator generates a signal which appears a fifth above the detected pitch).

## Examples

Here is an example of the *pvspitch* opcode. It uses the file *pvspitch.csd* [examples/pvspitch.csd]. This example uses realtime audio input but can be used for audiofile input as well.

### Example 777. Example of the *pvspitch* opcode

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o pvspitch.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 128
nchnls = 1

giwave ftgen 0, 0, 4096, 10, 1, 0.5, 0.333, 0.25, 0.2, 0.1666

instr 1

ifftsize = 1024
iwtype = 1 /* cleaner with hanning window */

a1 inch 1 ;Realtime audio input
;a1 soundin "input.wav" ;Use this line for file input

fsig pvsanal a1, ifftsize, ifftsize/4, ifftsize, iwtype
kfr, kamp pvspitch fsig, 0.01

adm oscil kamp, kfr * 1.5, giwave ;Generate note a fifth above detected pitch

out adm
endin

</CsInstruments>
<CsScore>

i 1 0 30

e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvsanal*, *pvsynth*, *pvsadsyn*, *pvscent*

## Credits

Author: Alan OCinneide

August 2005, added by Victor Lazzarini, August 2006

Part of the text has been adapted from the Csound Journal winter 2006 issue's article "Introducing PVSPITCH: A pitch tracking opcode for Csound" by Alan OCinneide. The article is available at: [www.csoundjournal.com/2006winter/pvspitch.html](http://www.csoundjournal.com/2006winter/pvspitch.html) [<http://www.csoundjournal.com/2006winter/pvspitch.html>]

# pvstanal

pvstanal — Phase vocoder analysis processing with onset detection/processing.

## Description

*pvstanal* implements phase vocoder analysis by reading function tables containing sampled-sound sources, with *GENOI*, and *pvstanal* will accept deferred allocation tables.

This opcode allows for time and frequency-independent scaling. Time is advanced internally, but controlled by a tempo scaling parameter; when an onset is detected, timescaling is momentarily stopped to avoid smearing of attacks. The quality of the effect is generally improved with phase locking switched on.

*pvstanal* will also scale pitch, independently of frequency, using a transposition factor (k-rate).

## Syntax

fsig **pvstanal** ktimescal, kamp, kpitch, ktab, [kdetect, kwrap, ioffset, ifftsize, ihop, idbthresh]

## Initialization

*ifftsize* -- FFT size (power-of-two), defaults to 2048.

*ihop* -- hopsize, defaults to 512

*ioffset* -- startup read offset into table, in secs.

*idbthresh* -- threshold for onset detection, based on dB power spectrum ratio between two successive windows. A detected ratio above it will cancel timescaling momentarily, to avoid smearing (defaults to 1). By default anything more than a 1 dB inter-frame power difference will be detected as an onset.

## Performance

*ktimescal* -- timescaling ratio, < 1 stretch, > 1 contract.

*kamp* -- amplitude scaling

*kpitch* -- grain pitch scaling (1=normal pitch, < 1 lower, > 1 higher; negative, backwards)

*kdetect* -- 0 or 1, to switch onset detection/processing. The onset detector checks for power difference between analysis windows. If more than what has been specified in the dbthresh parameter, an onset is declared. It suspends timescaling momentarily so the onsets are not modified.

*ktab* -- source signal function table. Deferred-allocation tables (see *GENOI*) are accepted, but the opcode expects a mono source. Tables can be switched at k-rate.

*kwrap* -- 0 or 1, to switch on/off table wrap-around read (default to 1)

## Examples

Here is an example of the *pvstanal* opcode. It uses the file *pvstanal.csd* [examples/pvstanal.csd].

**Example 778. Example of the pvstanal opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pvstanal.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gifil      ftgen      0, 0, 0, 1, "fox.wav", 0, 0, 1

instr 1

fsig       pvstanal  p4, 1, p5, gifil, p6, p7
aout       pvsynth   fsig
           outs      aout, aout

endin

instr 2

kspeed     randi      2, 2, 2 ;speed randomly between -2 and 2
kpitch     randi      2, 2, 2 ;pitch between 2 octaves lower or higher
fsig       pvstanal   kspeed, 1, octave(kpitch), gifil
aout       pvsynth    fsig
           outs      aout, aout

endin

</CsInstruments>
<CsScore>
;          speed pch det wrap
i 1 0 2.757 1 1 0 0
i 1 3 . 2 1 0 0
i 1 6 . 2 1 0 1
i 1 9 . 1 .75
i 2 12 10 ;random scratching
e
</CsScore>
</CsoundSynthesizer>

```

## Credits

Author: Victor Lazzarini  
February 2010

New plugin in version 5.13

February 2005.

# pvstencil

pvstencil — Transforms a pvoc stream according to a masking function table.

## Description

Transforms a pvoc stream according to a masking function table; if the pvoc stream amplitude falls below the value of the function for a specific pvoc channel, it applies a gain to that channel.

The pvoc stream amplitudes are compared to a masking table, if they fall below the table values, they are scaled by *kgain*. Prior to the operation, table values are scaled by *klevel*, which can be used as masking depth control.

Tables have to be at least  $\text{fftsize}/2$  in size; for most GENS it is important to use an extended-guard point (size power-of-two plus one), however this is not necessary with GEN43.

One of the typical uses of *pvstencil* would be in noise reduction. A noise print can be analysed with *pvanal* into a PVOC-EX file and loaded in a table with *GEN43*. This then can be used as the masking table for *pvstencil* and the amount of reduction would be controlled by *kgain*. Skipping post-normalisation will keep the original noise print average amplitudes. This would provide a good starting point for a successful noise reduction (so that *klevel* can be generally set to close to 1).

Other possible transformation effects are possible, such as filtering and 'inverse-masking'.

## Syntax

```
fsig pvstencil fsigin, kgain, klevel, iftable
```

## Performance

*fsig* -- output pv stream

*fsigin* -- input pv stream.

*kgain* -- 'stencil' gain.

*klevel* -- masking function level (scales the ftable prior to 'stenciling').

*iftable* -- masking function table.



### Warning

It is unsafe to use the same f-variable for both input and output of pvs opcodes. Using the same one might lead to undefined behavior on some opcodes. Use a different one on the left and right sides of the opcode.

## Examples

### Example 779. Example

```
fsig   pvsanal   asig, 1024, 256, 1024, 1
```

```
fclean pvstencil fsig, 0, 1, 1  
aclean pvsynth fclean
```

## Credits

Author: Victor Lazzarini  
November 2004

New plugin in version 5

November 2004.

# pvstrace

pvstrace — Retain only the N loudest bins.

## Description

Process a PV stream by retaining only the N bins with the highest amplitude, zeroing the others.

## Syntax

```
fsig pvstrace fsigin, kn
```

## Performance

*fsig* -- output pv stream

*fsigin* -- input pv stream

*kn* -- number of bins to be retained



### Warning

It is unsafe to use the same f-variable for both input and output of pvs opcodes. Using the same one might lead to undefined behavior on some opcodes. Use a different one on the left and right sides of the opcode.

## Examples

Here is an example of the use of the *pvstrace* opcode. It uses the file *pvstrace.csd* [examples/pvstrace.csd].

### Example 780. Example of the *pvstrace* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pvstrace.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kn = p4
asig diskin2 "beats.wav", 1
```

```
fsig pvsanal asig, 1024, 256, 1024, 1; analyse it
ftps pvstrace fsig, kn ; keep kn bins
atps pvsynth ftps ; synthesise it
outs      atps, atps

endin
</CsInstruments>
<CsScore>

i1 0 2 5
i1 + 2 10
i1 + 2 20
i1 + 2 40
i1 + 2 80
i1 + 2 160
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvsanal, pvsynth, pvsadsyn*

## Credits

Author: Victor Lazzarini  
2017

New plugin in version 6.09

## pvsvoc

pvsvoc — Combine the spectral envelope of one fsig with the excitation (frequencies) of another.

### Description

This opcode provides support for cross-synthesis of amplitudes and frequencies. It takes the amplitudes of one input fsig and combines with frequencies from another. It is a spectral version of the well-known channel vocoder.

### Syntax

```
fsig pvsvoc famp, fexc, kdepth, kgain [,kcoefs]
```

### Performance

*fsig* -- output pv stream

*famp* -- input pv stream from which the amplitudes will be extracted

*fexc* -- input pv stream from which the frequencies will be taken

*kdepth* -- depth of effect, affecting how much of the frequencies will be taken from the second fsig: 0, the output is the famp signal, 1 the output is the famp amplitudes and fexc frequencies.

*kgain* -- gain boost/attenuation applied to the output.

*kcoefs* -- number of cepstrum coefs used in spectral envelope estimation (defaults to 80).



#### Warning

It is unsafe to use the same f-variable for both input and output of pvs opcodes. Using the same one might lead to undefined behavior on some opcodes. Use a different one on the left and right sides of the opcode.

### Examples

Here is an example of the pvsvoc opcode. It uses the file *pvsvoc.csd* [examples/pvsvoc.csd].

#### Example 781. Example of the pvsvoc opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pvsvoc.wav -W ;; for file output any platform
</CsOptions>
```

```
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
odbfs = 1

gisaw ftgen 1, 0, 2048, 10, 1, 0.5, 0.3, 0.25, 0.2 ;sawtooth-like

instr 1

asig in ;get the signal in
asyn poscil .6, 150, gisaw ;excitation signal of 150 Hz

famp pvsanal asig, 1024, 256, 1024, 1 ;analyse in signal
fexc pvsanal asyn, 1024, 256, 1024, 1 ;analyse excitation signal
ftps pvsvoc famp, fexc, 1, 1 ;cross it
atps pvsynth ftps ;synthesise it
outs atps, atps

endin
</CsInstruments>
<CsScore>

i 1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

The example above shows a typical cross-synthesis operation. The input signal (say a vocal sound) is used for its amplitude spectrum. An oscillator with an arbitrary complex waveform produces the excitation signal, giving the vocal sound its pitch.

## Credits

Author: Victor Lazzarini  
April 2005

New plugin in version 5

April 2005.

# pvsynth

pvsynth — Resynthesise using a FFT overlap-add.

## Description

Resynthesise phase vocoder data (f-signal) using a FFT overlap-add.

## Syntax

```
ares pvsynth fsrc, [iinit]
```

## Performance

*ares* -- output audio signal

*fsrc* -- input signal

*iinit* -- not yet implemented.

## Examples

### Example 782. Example (using score-supplied f-table, assuming fsig *fftsize* = 1024)

```
; score f-table using cubic spline to define shaped peaks
f1 0 513 8 0 2 1 3 0 4 1 6 0 10 1 12 0 16 1 32 0 1 0 436 0

asig    buzz      20000, 199, 50, 1          ; pulswave source
fsig    pvsanal   asig, 1024, 256, 1024, 0   ; create fsig
kmod    linseg    0, p3/2, 1, p3/2, 0       ; simple control sig

fsigout pvsmaska fsig, 2, kmod             ; apply weird eq to fsig
aout    pvsynth   fsigout                  ; resynthesize,
        dispfft   aout, 0.1, 1024         ; and view the effect
```

Here is an example of the pvsynth opcode. It uses the file *pvsynth.csd* [examples/pvsynth.csd].

### Example 783. Example of the pvsynth opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o pvsynth.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 16
```

```
nchnls = 1
0dbfs = 1

;; example written by joachim heintz 2009

instr 1
ifftsize = 1024
ioverlap = ifftsize / 4
iwinsize = ifftsize
iwinshape = 1 ; von-Hann window
Sfile = "fox.wav"
ain soundin Sfile
fftin pvsanal ain, ifftsize, ioverlap, iwinsize, iwinshape; fft-analysis of the audio-signal
aout pvsynth fftin; resynthesis
out aout
endin

</CsInstruments>
<CsScore>
i 1 0 3
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvsanal, pvsadsyn*

## Credits

Author: Richard Dobson  
August 2001

New in version 4.13

February 2004. Thanks to a note from Francisco Vila, updated the example.

# pvs warp

pvs warp — Warp the spectral envelope of a PVS signal

## Description

Warp the spectral envelope of a PVS signal by means of shifting and scaling.

## Syntax

```
fsig pvs warp fsigin, kscal, kshift[, klowest, kmeth, kgain, kcoefs]
```

## Performance

*fsig* -- output pv stream

*fsigin* -- input pv stream

*kscal* -- spectral envelope scaling ratio. Values > 1 stretch the envelope and < 1 compress it.

*kshift* -- spectral envelope shift (in Hz), values > 0 shift the envelope linearly upwards and values < 0 shift it downwards.

*klowest* -- lowest frequency shifted (affects only *kshift*, defaults to 0).

*kmethod* -- spectral envelope extraction method 1: liltered cepstrum method; 2: true envelope method (defaults to 1).

*kgain* -- amplitude scaling (defaults to 1).

*kcoefs* -- number of cepstrum coefs used in formant preservation (defaults to 80).



### Warning

It is unsafe to use the same *f*-variable for both input and output of pvs opcodes. Using the same one might lead to undefined behavior on some opcodes. Use a different one on the left and right sides of the opcode.

## Examples

Here is an example of the pvs warp opcode. It uses the file *pvs warp.csd* [examples/pvs warp.csd].

### Example 784. Example of the pvs warp opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
```

```
; For Non-realtime ouput leave only the line below:
; -o pvswarp.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kscal = p4
kshift = p5
asig soundin "fox.wav" ; get the signal in
fsig pvsanal asig, 1024, 256, 1024, 1 ; analyse it
ftps pvswarp fsig, kscal, kshift ; warp it
atps pvsynth ftps ; synthesise it
outs atps/2, atps/2

endin
</CsInstruments>
<CsScore>
;change scale
i 1 0 3 1
i 1 + . 1.5
i 1 + . 3
i 1 + . .25
;change shift
i 1 + . 1 0
i 1 + . . 300
i 1 + . . 0
i 1 + . . -300
e
</CsScore>
</CsoundSynthesizer>
```

Used with vocal sounds, it will shift the formants and result in a changed vowel timbre, similar to the effect of a singer inhaling helium (the 'donald duck' effect).

## See Also

*pvsanal, pvsynth, pvsadsyn*

## Credits

Author: Victor Lazzarini  
November 2004

New plugin in version 5

November 2004.

# pvs2tab

pvs2tab — Copies spectral data to k-rate arrays (or t-variables). Also known as pvs2array.

## Description

Copies a pvs frame to a t-variable. Currently only AMP+FREQ and AMP+PHASE formats allowed. Since t-vars are an earlier version of k-rate arrays, the opcode also works with these. The opcode pvs2array is an alias of this one.

## Syntax

```
kframe pvs2tab tvar|kvar[], fsig  
kframe pvs2tab kmags[], kfreqs[], fsig
```

## Performance

*kframe* -- current copied frame number. It can be used to detect when a new frame has been copied.

*tvar/kvar[]* -- t-variable or k-rate array containing the output. It is produced at every k-period, but may not contain a new frame, pvs frames are produced at their own frame rate that is independent of kr. Generally, this vector needs to be big enough to contain the frame samples, i.e.  $N+2$  ( $N$  is the dft size). If smaller, only a portion of the frame will be copied; if bigger, unused points will exist at higher indexes.

*kmags[], kfreqs[]* k-rate arrays containing magnitude and frequency values. These arrays need to be generally be of size  $N/2 + 1$ , where  $N$  is dft frame size. If smaller, only a portion of the pvs frame will be copied; if bigger, unused points will exist at higher indexes.

*fsig* -- input fsig to be copied.

## Examples

### Example 785. Example

```
karr[] init 1026  
a1 inch 1  
fsigl pvsanal a1, 1024,256,1024, 1  
kframe pvs2tabkarr, fsigl
```

## Credits

Author: Victor Lazzarini  
October 2011

New plugin in version 5

October 2011.

## pyassign Opcodes

`pyassign` — Assign the value of the given Csound variable to a Python variable possibly destroying its previous content.

### Syntax

```
pyassign "variable", kvalue  
pyassigni "variable", ivalue  
pylassign "variable", kvalue  
pylassigni "variable", ivalue  
pyassignt ktrigger, "variable", kvalue  
pylassignt ktrigger, "variable", kvalue
```

### Description

Assign the value of the given Csound variable to a Python variable possibly destroying its previous content. The resulting Python object will be a float.

### Credits

Copyright (c) 2002 by Maurizio Umberto Puxeddu. All rights reserved. Portions copyright (c) 2004 and 2005 by Michael Gogins. This document has been updated Sunday 25 July 2004 and 1 February 2005 by Michael Gogins.

# pycall Opcodes

`pycall` — Invoke the specified Python callable at k-time and i-time (i suffix), passing the given arguments. The call is performed in the global environment, and the result (the returning value) is copied into the Csound output variables specified.

## Syntax

	<code>pycall</code>	"callable", karg1, ...
<code>kresult</code>	<code>pycall11</code>	"callable", karg1, ...
<code>kresult1, kresult2</code>	<code>pycall12</code>	"callable", karg1, ...
<code>kr1, kr2, kr3</code>	<code>pycall13</code>	"callable", karg1, ...
<code>kr1, kr2, kr3, kr4</code>	<code>pycall14</code>	"callable", karg1, ...
<code>kr1, kr2, kr3, kr4, kr5</code>	<code>pycall15</code>	"callable", karg1, ...
<code>kr1, kr2, kr3, kr4, kr5, kr6</code>	<code>pycall16</code>	"callable", karg1, ...
<code>kr1, kr2, kr3, kr4, kr5, kr6, kr7</code>	<code>pycall17</code>	"callable", karg1, ...
<code>kr1, kr2, kr3, kr4, kr5, kr6, kr7, kr8</code>	<code>pycall18</code>	"callable", karg1, ...
	<code>pycallt</code>	ktrigger, "callable", karg1, ...
<code>kresult</code>	<code>pycall11t</code>	ktrigger, "callable", karg1, ...
<code>kresult1, kresult2</code>	<code>pycall12t</code>	ktrigger, "callable", karg1, ...
<code>kr1, kr2, kr3</code>	<code>pycall13t</code>	ktrigger, "callable", karg1, ...
<code>kr1, kr2, kr3, kr4</code>	<code>pycall14t</code>	ktrigger, "callable", karg1, ...
<code>kr1, kr2, kr3, kr4, kr5</code>	<code>pycall15t</code>	ktrigger, "callable", karg1, ...
<code>kr1, kr2, kr3, kr4, kr5, kr6</code>	<code>pycall16t</code>	ktrigger, "callable", karg1, ...
<code>kr1, kr2, kr3, kr4, kr5, kr6, kr7</code>	<code>pycall17t</code>	ktrigger, "callable", karg1, ...
<code>kr1, kr2, kr3, kr4, kr5, kr6, kr7, kr8</code>	<code>pycall18t</code>	ktrigger, "callable", karg1, ...
	<code>pycalli</code>	"callable", karg1, ...
<code>irestult</code>	<code>pycall11i</code>	"callable", iarg1, ...
<code>irestult1, irestult2</code>	<code>pycall12i</code>	"callable", iarg1, ...
<code>ir1, ir2, ir3</code>	<code>pycall13i</code>	"callable", iarg1, ...
<code>ir1, ir2, ir3, ir4</code>	<code>pycall14i</code>	"callable", iarg1, ...
<code>ir1, ir2, ir3, ir4, ir5</code>	<code>pycall15i</code>	"callable", iarg1, ...
<code>ir1, ir2, ir3, ir4, ir5, ir6</code>	<code>pycall16i</code>	"callable", iarg1, ...
<code>ir1, ir2, ir3, ir4, ir5, ir6, ir7</code>	<code>pycall17i</code>	"callable", iarg1, ...
<code>ir1, ir2, ir3, ir4, ir5, ir6, ir7, ir8</code>	<code>pycall18i</code>	"callable", iarg1, ...
<code>pycalln</code>	"callable", nresults, kresult1, ..., kresultn, karg1, ...	
<code>pycallni</code>	"callable", nresults, irestult1, ..., irestultn, iarg1, ...	
	<code>pylcall</code>	"callable", karg1, ...

kresult	<b>pylcall1</b>	"callable", karg1, ...
kresult1, kresult2	<b>pylcall2</b>	"callable", karg1, ...
kr1, kr2, kr3	<b>pylcall3</b>	"callable", karg1, ...
kr1, kr2, kr3, kr4	<b>pylcall4</b>	"callable", karg1, ...
kr1, kr2, kr3, kr4, kr5	<b>pylcall5</b>	"callable", karg1, ...
kr1, kr2, kr3, kr4, kr5, kr6	<b>pylcall6</b>	"callable", karg1, ...
kr1, kr2, kr3, kr4, kr5, kr6, kr7	<b>pylcall7</b>	"callable", karg1, ...
kr1, kr2, kr3, kr4, kr5, kr6, kr7, kr8	<b>pylcall8</b>	"callable", karg1, ...
	<b>pylcallt</b>	ktrigger, "callable", karg1, ...
kresult	<b>pylcall1t</b>	ktrigger, "callable", karg1, ...
kresult1, kresult2	<b>pylcall2t</b>	ktrigger, "callable", karg1, ...
kr1, kr2, kr3	<b>pylcall3t</b>	ktrigger, "callable", karg1, ...
kr1, kr2, kr3, kr4	<b>pylcall4t</b>	ktrigger, "callable", karg1, ...
kr1, kr2, kr3, kr4, kr5	<b>pylcall5t</b>	ktrigger, "callable", karg1, ...
kr1, kr2, kr3, kr4, kr5, kr6	<b>pylcall6t</b>	ktrigger, "callable", karg1, ...
kr1, kr2, kr3, kr4, kr5, kr6, kr7	<b>pylcall7t</b>	ktrigger, "callable", karg1, ...
kr1, kr2, kr3, kr4, kr5, kr6, kr7, kr8	<b>pylcall8t</b>	ktrigger, "callable", karg1, ...
	<b>pylcalli</b>	"callable", karg1, ...
	<b>pylcall1i</b>	"callable", iarg1, ...
	<b>pylcall2i</b>	"callable", iarg1, ...
	<b>pylcall3i</b>	"callable", iarg1, ...
	<b>pylcall4i</b>	"callable", iarg1, ...
	<b>pylcall5i</b>	"callable", iarg1, ...
	<b>pylcall6i</b>	"callable", iarg1, ...
	<b>pylcall7i</b>	"callable", iarg1, ...
	<b>pylcall8i</b>	"callable", iarg1, ...
<b>pylcalln</b>	"callable", nresults, kresult1, ..., kresultn, karg1, ...	
<b>pylcallni</b>	"callable", nresults, ireresult1, ..., ireresultn, iarg1, ...	

## Description

This family of opcodes call the specified Python callable at k-time and i-time (i suffix), passing the given arguments. The call is performed in the global environment and the result (the returning value) is copied into the Csound output variables specified.

They pass any number of parameters which are cast to float inside the Python interpreter.

The *pycall/pycalli*, *pycall1/pycall1i* ... *pycall8/pycall8i* opcodes can accommodate for a number of results ranging from 0 to 8 according to their numerical prefix (0 is omitted).

The *pycalln/pycallni* opcodes can accommodate for any number of results: the callable name is followed by the number of output arguments, then come the list of Csound output variable and the list of parameters to be passed.

The returning value of the callable must be `None` for `pycall` or `pycalli`, a float for `pycall1i` or `pycall1i` and a tuple (with proper size) of floats for the `pycall2/pycall2i ... pycall8/pycall8i` and `pycalln/pycallni` opcodes.

## Examples

### Example 786. Calling a C or Python function

Supposing we have previously defined or imported a function named `get_number_from_pool` as:

```
from random import random, choice

# a pool of 100 numbers
pool = [i ** 1.3 for i in range(100)]

def get_number_from_pool(n, p):
    # substitute an old number with the new number?
    if random() < p:
        i = choice(range(len(pool)))
        pool[i] = n

    # return a random number from the pool
    return choice(pool)
```

then the following orchestra code

```
k2 pycall1 "get_number_from_pool", k1, p6
```

would set `k2` randomly from a pool of numbers changing in time. You can pass new pools elements and control the change rate from the orchestra.

### Example 787. Calling a Function Object

A more generic implementation of the previous example makes use of a simple function object:

```
from random import random, choice

class GetNumberFromPool:
    def __init__(self, e, begin=0, end=100, step=1):
        self.pool = [i ** e for i in range(begin, end, step)]

    def __call__(self, n, p):
        # substitute an old number with the new number?
        if random() < p:
            i = choice(range(len(pool)))
            pool[i] = n

        # return a random number from the pool
        return choice(pool)

get_number_from_pool1 = GetNumberFromPool(1.3)
get_number_from_pool2 = GetNumberFromPool(1.5, 50, 250, 2)
```

Then the following orchestra code:

```
k2 pycall1 "get_number_from_pool1", k1, p6
k4 pycall1 "get_number_from_pool2", k3, p7
```

would set `k2` and `k4` randomly from a pool of numbers changing in time. You can pass new pools elements (here `k1` and `k3`) and control the change rate (here `p6` and `p7`) from the orchestra.

As you can see in the first snippet, you can customize the initialization of the pool as well as create several pools.

## Credits

Copyright (c) 2002 by Maurizio Umberto Puxeddu. All rights reserved. Portions copyright (c) 2004 and 2005 by Michael Gogins. This document has been updated Sunday 25 July 2004 and 1 February 2005 by Michael Gogins.

## pyeval Opcodes

`pyeval` — Evaluate a generic Python expression and store the result in a Csound variable at k-time or i-time (i suffix).

### Syntax

```
kresult pyeval "expression"  
iresult pyevali "expression"  
kresult pyleval "expression"  
iresult pylevali "expression"  
kresult pyevalt ktrigger, "expression"  
kresult pylevalt ktrigger, "expression"
```

### Description

These opcodes evaluate a generic Python expression and store the result in a Csound variable at k-time or i-time (i suffix).

The expression must evaluate in a float or an object that can be cast to a float.

They can be used effectively to transfer data from a Python object into a Csound variable.

### Example of the pyleval Opcode Group

The code:

```
k1          pyeval      "v1"
```

will copy the content of the Python variable `v1` into the Csound variable `k1` at each k-time.

### Credits

Copyright (c) 2002 by Maurizio Umberto Puxeddu. All rights reserved. Portions copyright (c) 2004 and 2005 by Michael Gogins. This document has been updated Sunday 25 July 2004 and 1 February 2005 by Michael Gogins.

# pyexec Opcodes

`pyexec` — Execute a script from a file at k-time or i-time (i suffix).

## Syntax

```
pyexec "filename"  
pyexeci "filename"  
pylexec "filename"  
pylexeci "filename"  
pyexec ktrigger, "filename"  
plyexec ktrigger, "filename"
```

## Description

Execute a script from a file at k-time or i-time (i suffix).

This is not the same as calling the script with the `system()` call, since the code is executed by the embedded interpreter.

The code contained in the specified file is executed in the global environment for opcodes `pyexec` and `pyexeci` and in the private environment for the opcodes `pylexec` and `pylexeci`.

These opcodes perform no message passing. However, since the statement has access to the main namespace and the private namespace, it can interact with objects previously created in that environment.

The "local" version of the *pyexec* opcodes are useful when the code ran by different instances of an instrument should not interact.

## Example of the pyexec Opcode Group

### Example 788. Orchestra (pyexec.orc)

```
sr=44100  
kr=4410  
ksmps=10  
nchnls=1  
  
;If you're not running CsoundAC you need the following line  
;to initialize the python interpreter  
;pyinit  
  
pyruni "import random"  
  
pyexeci "pyexec1.py"  
  
instr 1  
  
pyexec "pyexec2.py"  
  
pylexeci "pyexec3.py"  
pylexec "pyexec4.py"
```

```
endin
```

### Example 789. Score (pyexec.sco)

```
i1 0 0.01
i1 0 0.01
```

### Example 790. The pyexec1.py Script

```
import time, os

print
print "Welcome to Csound!"

try:
    s = ', %s?' % os.getenv('USER')
except:
    s = '?'

print 'What sound do you want to hear today%s' % s
answer = raw_input()
```

### Example 791. The pyexec2.py script

```
print 'your answer is "%s"' % answer
```

### Example 792. The pyexec3.py script

```
message = 'a private random number: %f' % random.random()
```

### Example 793. The pyexec4.py script

```
print message
```

If I run this example on my machine I get something like:

```
Using ../../csound.xmg
Csound Version 4.19 (Mar 23 2002)
Embedded Python interpreter version 2.2
orchname: pyexec.orc
scorename: pyexec.sco
sorting score ...
... done
orch compiler:
11 lines read
    instr 1
Csound Version 4.19 (Mar 23 2002)
displays suppressed

Welcome to Csound!
What sound do you want to hear today, maurizio?
```

then I answer

a sound

then Csound continues with the normal performance

```
your answer is "a sound"  
a private random number: 0.884006  
new alloc for instr 1:  
your answer is "a sound"  
a private random number: 0.884006  
your answer is "a sound"  
a private random number: 0.889868  
your answer is "a sound"  
a private random number: 0.884006  
your answer is "a sound"  
a private random number: 0.889868  
your answer is "a sound"  
a private random number: 0.884006  
your answer is "a sound"  
...
```

In the same instrument a message is created in the private namespace and printed, appearing different for each instance.

## Credits

Copyright (c) 2002 by Maurizio Umberto Puxeddu. All rights reserved. Portions copyright (c) 2004 and 2005 by Michael Gogins. This document has been updated Sunday 25 July 2004 and 1 February 2005 by Michael Gogins.

# pyinit Opcodes

`pyinit` — Initialize the Python interpreter.

## Syntax

`pyinit`

## Description

In Csound, you must first invoke the `pyinit` opcode in the orchestra header to initialize the Python interpreter, before using any of the other Python opcodes.

But if you use the Python opcodes within CsoundAC, or from a python frontend using the `csnd6` module, Csound, you need not invoke `pyinit`, because the Python interpreter will have already been initialized. In this case, CsoundAC (or the `csnd6` python module) automatically creates a Python interface to the Csound API. In CsoundAC, this exists in the form a global instance of the `CsoundAC.CppSound` class named `csound`. From a python frontend which imports the `csnd6` module, the name of the variable holding the Csound instance will depend on the frontend code. Therefore, Python code written in the Csound orchestra has access to the global `csound` object.

The running Csound instance in which `pyinit` has been called is stored in a global python variable called `_CSOUND_`. This holds the actual memory address of the instance and it can be used with the `csnd6` module via the `csoundGetInstance(instance)` call. This python function returns an object that can be used with all the Csound API functions.

## Credits

Copyright (c) 2002 by Maurizio Umberto Puxeddu. All rights reserved. Portions copyright (c) 2004 and 2005 by Michael Gogins, and (c) 2013, V Lazzarini.

# pyrun Opcodes

pyrun — Run a Python statement or block of statements.

## Syntax

```
pyrun "statement"  
pyruni "statement"  
pylrun "statement"  
pylruni "statement"  
pyrunt ktrigger, "statement"  
pylrunt ktrigger, "statement"
```

## Description

Execute the specified Python statement at k-time (*pyrun* and *pylrun*) or i-time (*pyruni* and *pylruni*).

The statement is executed in the global environment for *pyrun* and *pyruni* or the local environment for *pylrun* and *pylruni*.

These opcodes perform no message passing. However, since the statement have access to the main namespace and the private namespace, it can interact with objects previously created in that environment.

The "local" version of the *pyrun* opcodes are useful when the code ran by different instances of an instrument should not interact.

## Example of the pyrun Opcode Group

### Example 794. Orchestra

```
sr=44100  
kr=4410  
ksmps=10  
nchnls=1  
  
;If you're not running CsoundAC you need the following line  
;to initialize the python interpreter  
;pyinit  
  
pyruni "import random"  
  
instr 1  
    ; This message is stored in the main namespace  
    ; and is the same for every instance  
    pyruni "message = 'a global random number: %f' % random.random()"  
    pyrun "print message"  
  
    ; This message is stored in the private namespace  
    ; and is different for different instances  
    pylruni "message = 'a private random number: %f' % random.random()"  
    pylrun "print message"  
endin
```

## Example 795. Score

```
i1 0 0.1
```

Running this score you should get intermixed pairs of messages from the two instances of instrument 1.

The first message of each pair is stored into the main namespace and so the second instance overwrites the message of the first instance. The result is that first message will be the same for both instances.

The second message is different for the two instances, being stored in the private namespace.

## Credits

Copyright (c) 2002 by Maurizio Umberto Puxeddu. All rights reserved. Portions copyright (c) 2004 and 2005 by Michael Gogins. This document has been updated Sunday 25 July 2004 and 1 February 2005 by Michael Gogins.

# pwd

`pwd` — Asks the underlying operating system for the current directory name as a string.

## Description

`pwd` call the operating system to determine the current directory (folder). `pwd` runs at i-time only.

## Syntax

```
Sres pwd
```

## Performance

`Sres --` the returned string.

## Example

Here is an example of the `pwd` opcode. It uses the file `pwd.csd` [examples/pwd.csd].

### Example 796. Example of the `pwd` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out      Audio in
-odac            ; -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o system.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 1
; Waits for command to execute before continuing
Swd pwd
puts Swd, 1
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for minimal time.
i 1 0 0.1
e

</CsScore>
```

`</CsoundSynthesizer>`

## Credits

Author: John ffitch  
July 2012

New in version 5.18

# qinf

qinf — Questions whether the argument is a infinite number

## Description

Returns the number of times the argument is not a number, with the sign of the first infinity.

## Syntax

`qinf(x)` (no rate restriction)

## Examples

Here is an example of the qinf opcode. It uses the file `qinf.csd` [examples/qinf.csd].

### Example 797. Example of the qinf opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
--rtaudio=alsa -o dac:hw:0
</CsOptions>
<CsInstruments>
nchnls = 2
ksmps = 400

#define WII_B          #3#
#define WII_A          #4#
#define WII_R_A        #304#
#define WII_PITCH      #20#
#define WII_ROLL       #21#

gkcnt init 1

instr 1
  i1 wiiconnect 3,1

      wiirange $WII_PITCH., -20, 0
  kt wiidata $WII_B.
  ka wiidata $WII_A.
  kra wiidata $WII_R_A.
  gka wiidata $WII_PITCH.
  gkp wiidata $WII_ROLL.
; If the B (trigger) button is pressed then activate a note
  if (kt==0) goto ee
  if (qinf(gka)) goto ee
  if (qinf(gkp)) goto ee
  event "i", 2, 0, 5
  gkcnt = gkcnt + 1
  printk2 kb
endin

instr 2
  a1 oscil ampdbfs(gka), 440+gkp, 1
  outs a1, a1
```

```
    endin  
  
  </CsInstruments>  
  
  <CsScore>  
    f1 0 4096 10 1  
    i1 0 300  
  
  </CsScore>  
  
</CsoundSynthesizer>
```

## Credits

Written by John ffitth.

New in Csound 5.14

# qnan

qnan — Questions whether the argument is not a number

## Description

Returns the number of times the argument is not a number.

## Syntax

`qnan(x)` (no rate restriction)

## Examples

Here is an example of the qnan opcode. It uses the file `qnan.csd` [examples/qnan.csd].

### Example 798. Example of the qnan opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
--rtaudio=alsa -o dac:hw:0
</CsOptions>
<CsInstruments>
nchnls = 2
ksmps = 400

#define WII_B          #3#
#define WII_A          #4#
#define WII_R_A        #304#
#define WII_PITCH      #20#
#define WII_ROLL       #21#

gkcnt init 1

instr 1
  i1 wiiconnect 3,1

      wiirange $WII_PITCH., -20, 0
  kt wiidata $WII_B.
  ka wiidata $WII_A.
  kra wiidata $WII_R_A.
  gka wiidata $WII_PITCH.
  gkp wiidata $WII_ROLL.
; If the B (trigger) button is pressed then activate a note
  if (kt==0) goto ee
  if (qnan(gka)) goto ee
  if (qnan(gkp)) goto ee
  event "i", 2, 0, 5
  gkcnt = gkcnt + 1
  printk2 kb
endin

instr 2
  a1 oscil ampdbfs(gka), 440+gkp, 1
  outs a1, a1
```

```
    endin  
  
    </CsInstruments>  
  
    <CsScore>  
    f1 0 4096 10 1  
    i1 0 300  
  
    </CsScore>  
  
    </CsoundSynthesizer>
```

## Credits

Written by John fitch.

New in Csound 5.14

## r2c

r2c — Real to complex format conversion.

### Description

Converts a real-valued input array in real-imaginary interleaved complex format, setting its imaginary parts to 0. The output array will be double the size of the input. This is a utility operation to facilitate complex-value operations on real arrays.

### Syntax

```
kout[] r2c kin[]
```

### Performance

*kout[]* -- output array containing the complex-valued real-imaginary output. It will be created if it does not exist.

*kin[]* -- input array containing the real-valued input.

### Examples

Here is an example of the r2c opcode. It uses the file *r2c.csd* [examplesr2c.csd].

#### Example 799. Example of the r2c opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsoundOptions>
-d -o dac
</CsoundOptions>
<CsoundInstruments>
ksmps = 64

instr 1
  ifftsize = 1024
  kcnt init 0
  kIn[] init ifftsize
  kOut[] init ifftsize

  a1 oscili 0dbfs/2, 440

  if kcnt >= ifftsize then
    kCmplx[] r2c kIn
    kSpec[] fft kCmplx
    kCmplx ifft kSpec
    kOut c2r kCmplx
    kcnt = 0
  endif

  kIn[] shiftin a1
  a2 shiftout kOut
```

```
kcnt += ksmps
  out a2
endin
</CsInstruments>
<CsScore>
i1 0 10
</CsScore>
</CsoundSynthesizer>
```

## See Also

Vectorial opcodes, array opcodes

## Credits

Author: Victor Lazzarini  
NUI Maynooth  
2014

New in version 6.04

# rand

rand — Generates a controlled random number series.

## Description

Output is a controlled random number series between *-amp* and *+amp*

## Syntax

```
ares rand xamp [, iseed] [, isel] [, ioffset]
kres rand xamp [, iseed] [, isel] [, ioffset]
```

## Initialization

*iseed* (optional, default=0.5) -- a seed value for the recursive pseudo-random formula. A value between 0 and 1 will produce an initial output of *kamp \* iseed*. A value greater than 1 will be seeded from the system clock. A negative value will cause seed re-initialization to be skipped. The default seed value is .5.

*isel* (optional, default=0) -- if zero, a 16-bit number is generated. If non-zero, a 31-bit random number is generated. Default is 0.

*ioffset* (optional, default=0) -- a base value added to the random result. New in Csound version 4.03.

## Performance

*kamp*, *xamp* -- range over which random numbers are distributed.

*ares*, *kres* -- Random number produced.

The internal pseudo-random formula produces values which are uniformly distributed over the range *kamp* to *-kamp*. *rand* will thus generate uniform white noise with an R.M.S value of *kamp / root 2*.

The value *ares* or *kres* is within a half-closed interval which contains *-xamp*, but never contains *+xamp*.

## Examples

Here is an example of the rand opcode. It uses the file *rand.csd* [examples/rand.csd].

### Example 800. Example of the rand opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o rand.wav -W ;; for file output any platform
```

```
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;same values every time

krnd rand 100
  printk .5, krnd ; look
aout oscili 0.8, 440+krnd, 1 ; & listen
  outs aout, aout

endin

instr 2 ;different values every time

krnd rand 100, 10 ; seed from system clock
  printk .5, krnd ; look
aout oscili 0.8, 440+krnd, 1 ; & listen
  outs aout, aout

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine wave.

i 1 0 1
i 2 2 1
e

</CsScore>
</CsoundSynthesizer>
```

The example will produce the following output:

```
i 1 1 time 0.00067: 50.00305
i 1 1 time 0.50000: 62.71362
i 1 1 time 1.00000: -89.31885

WARNING: Seeding from current time 472230558

i 2 2 time 2.00067: -70.65735
i 2 2 time 2.50000: 69.15283
i 2 2 time 3.00000: -48.79761
```

## See Also

*randh, randi*

## Credits

Thanks to a note from John ffitich, I changed the names of the parameters.

# randh

randh — Generates random numbers and holds them for a period of time.

## Description

Generates random numbers and holds them for a period of time.

## Syntax

```
ares randh xamp, xcps [, iseed] [, isize] [, ioffset]
kres randh kamp, kcps [, iseed] [, isize] [, ioffset]
```

## Initialization

*iseed* (optional, default=0.5) -- seed value for the recursive pseudo-random formula. A value between 0 and +1 will produce an initial output of  $kamp * iseed$ . A negative value will cause seed re-initialization to be skipped. A value greater than 1 will seed from system time, this is the best option to generate a different random sequence for each run.

*isize* (optional, default=0) -- if zero, a 16 bit number is generated. If non-zero, a 31-bit random number is generated. Default is 0.

*ioffset* (optional, default=0) -- a base value added to the random result. New in Csound version 4.03.

## Performance

*kamp*, *xamp* -- range over which random numbers are distributed.

*kcps*, *xcps* -- the frequency which new random numbers are generated.

The internal pseudo-random formula produces values which are uniformly distributed over the range  $-kamp$  to  $+kamp$ . *rand* will thus generate uniform white noise with an R.M.S value of  $kamp / \text{root } 2$ .

The remaining units produce band-limited noise: the *kcps* and *xcps* parameters permit the user to specify that new random numbers are to be generated at a rate less than the sampling or control frequencies. *randh* will hold each new number for the period of the specified cycle.

## Examples

Here is an example of the randh opcode. It uses the file *randh.csd* [examples/randh.csd].

### Example 801. Example of the randh opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o randh.wav -W ;; for file output any platform
```

```
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;same values every time

krnd randh 100, 10
  printk2 krnd ; look
aout oscili 0.8, 440+krnd, 1 ; & listen
  outs aout, aout

endin

instr 2 ;different values every time

krnd randh 100, 10, 10 ; seed from system clock
  printk2 krnd ; look
aout oscili 0.8, 440+krnd, 1 ; & listen
  outs aout, aout

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine wave.

i 1 0 1
i 2 2 1
e
</CsScore>
</CsoundSynthesizer>
```

The example will produce the following output:

```
i1 50.00000
i1 50.00305
i1 97.68677
i1 -44.25354
i1 -61.56006
i1 -75.91248
i1 67.57202
i1 12.83875
i1 5.39551
i1 -95.18738
```

WARNING: Seeding from current time 684387922

```
i2 -13.81226
i2 -16.49475
i2 69.51904
i2 35.04944
i2 47.47925
i2 63.25378
i2 -59.61914
i2 50.93079
i2 -6.46362
i2 5.89294
```

## See Also

*rand*, *randi*

# randi

rand — Generates a controlled random number series with interpolation between each new number.

## Description

Generates a controlled random number series with interpolation between each new number.

## Syntax

```
ares randi xamp, xcps [, iseed] [, isize] [, ioffset]
kres randi kamp, kcps [, iseed] [, isize] [, ioffset]
```

## Initialization

*iseed* (optional, default=0.5) -- seed value for the recursive pseudo-random formula. A value between 0 and +1 will produce an initial output of *kamp* \* *iseed*. A negative value will cause seed re-initialization to be skipped. A value greater than 1 will seed from system time, this is the best option to generate a different random sequence for each run.

*isize* (optional, default=0) -- if zero, a 16 bit number is generated. If non-zero, a 31-bit random number is generated. Default is 0.

*ioffset* (optional, default=0) -- a base value added to the random result. New in Csound version 4.03.

## Performance

*kamp*, *xamp* -- range over which random numbers are distributed.

*kcps*, *xcps* -- the frequency which new random numbers are generated.

The internal pseudo-random formula produces values which are uniformly distributed over the range *kamp* to *-kamp*. *rand* will thus generate uniform white noise with an R.M.S value of *kamp* / *root 2*.

The remaining units produce band-limited noise: the *kcps* and *xcps* parameters permit the user to specify that new random numbers are to be generated at a rate less than the sampling or control frequencies. *randi* will produce straight-line interpolation between each new number and the next.

## Examples

Here is an example of the *randi* opcode. It uses the file *randi.csd* [examples/rand\_i.csd].

### Example 802. Example of the *randi* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;;realtime audio out
```

```

;-iadc    ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o randi.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;same values every time

krnd randi 100, 10
  printk .5, krnd ; look
aout oscili 0.8, 440+krnd, 1 ; & listen
  outs aout, aout

endin

instr 2 ;different values every time

krnd randi 100, 10, 10 ; seed from system clock
  printk .5, krnd ; look
aout oscili 0.8, 440+krnd, 1 ; & listen
  outs aout, aout

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine wave.

i 1 0 1
i 2 2 1
e
</CsScore>
</CsoundSynthesizer>

```

The example will produce the following output:

```

i 1 1 time      0.00067:    50.00000
i 1 1 time      0.50000:   -75.81672
i 1 1 time      1.00000:    95.93833

WARNING: Seeding from current time 1482746120

i 2 2 time      2.00067:   -17.94434
i 2 2 time      2.50000:   -14.11875
i 2 2 time      3.00000:   -72.41545

```

## See Also

*rand*, *randh*

# random

random — Generates a controlled pseudo-random number series between min and max values.

## Description

Generates is a controlled pseudo-random number series between min and max values.

## Syntax

```
ares random kmin, kmax
ires random imin, imax
kres random kmin, kmax
```

## Initialization

*imin* -- minimum range limit

*imax* -- maximum range limit

## Performance

*kmin* -- minimum range limit

*kmax* -- maximum range limit

The *random* opcode is similar to *linrand* and *trirand* but sometimes I [Gabriel Maldonado] find it more convenient because allows the user to set arbitrary minimum and maximum values.

## Examples

Here is an example of the random opcode. It uses the file *random.csd* [examples/random.csd].

### Example 803. Example of the random opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o random.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
```

```
instr 1 ;same values every time

krnd random 100, 1000
  printk .5, krnd ; look
aout oscili 0.8, 440+krnd, 1 ; & listen
  outs aout, aout

endin

instr 2 ;different values every time

seed 0
krnd random 100, 1000 ; seed from system clock
  printk .5, krnd ; look
aout oscili 0.8, 440+krnd, 1 ; & listen
  outs aout, aout

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine wave.

i 1 0 1
i 2 2 1
e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like:

```
i 1 time 0.00067: 894.58566
i 1 time 0.50000: 748.44281
i 1 time 1.00000: 328.29916
```

WARNING: Seeding from current time 1656666052

```
i 2 time 2.00067: 690.71466
i 2 time 2.50000: 459.42445
i 2 time 3.00000: 100.85594
```

## See Also

*linrand, randomh, randomi, trirand*

## Credits

Author: Gabriel Maldonado

# randomh

randomh — Generates random numbers with a user-defined limit and holds them for a period of time.

## Description

Generates random numbers with a user-defined limit and holds them for a period of time.

## Syntax

```
ares randomh kmin, kmax, xcps [,imode] [,ifirstval]
```

```
kres randomh kmin, kmax, kcps [,imode] [,ifirstval]
```

## Initialization

*imode* (optional, default=0) -- generation mode of the first output value (see below)

*ifirstval* (optional, default=0) -- first output value

## Performance

*kmin* -- minimum range limit

*kmax* -- maximum range limit

*kcps*, *xcps* -- rate of random break-point generation

The *randomh* opcode is similar to *randh* but allows the user to set arbitrary minimum and maximum values.

When *imode* = 0 (the default), the *kmin* argument value is outputted during  $1/kcps$  (resp.  $1/xcps$ ) seconds at the beginning of the note. Then, the normal process takes place with a new random number generated and held every  $1/kcps$  (resp.  $1/xcps$ ) seconds.

When *imode* = 2, the *ifirstval* argument value is outputted during  $1/kcps$  (resp.  $1/xcps$ ) seconds at the beginning of the note. Then, the normal process takes place with a new random number generated and held every  $1/kcps$  (resp.  $1/xcps$ ) seconds.

When *imode* = 3, the generation process begins with a random value from the initialization time.

## Examples

Here is an example of the *randomh* opcode. It uses the file *randomh.csd* [examples/randomh.csd].

### Example 804. Example of the randomh opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>
```

```

; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o randomh.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

seed 0

; Instrument #1.
instr 1
; Choose a random frequency between 220 and 440 Hz.
; Generate new random numbers at 10 Hz.
kmin  init 220
kmax  init 440
kcps  init 10
imode  =    p4
ifstval =  p5

printf_i "\nMode: %d\n", 1, imode
k1 randomh kmin, kmax, kcps, imode, ifstval
printk2 k1
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second,
; each time with a different mode.
i 1 0 1
i 1 1 1 2 330
i 1 2 1 3
e

</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like this:

```

Mode: 0
i1 220.00000
i1 396.26079
i1 240.75446
i1 364.24577
...

Mode: 2
i1 330.00000
i1 416.50935
i1 356.11619
i1 433.59324
...

Mode: 3
i1 261.17741
i1 402.00891
i1 393.86592
i1 307.19839
...

```

## See Also

*randh, random, randomi*

## Credits

Author: Gabriel Maldonado

Arguments *imode* and *ifirstval* added by François Pinot, Jan. 2011, after a discussion with Peiman Khosravi on the csnd list.

Example written by Kevin Conder, adapted for new args by François Pinot.

# randomi

`randomi` — Generates a user-controlled random number series with interpolation between each new number.

## Description

Generates a user-controlled random number series with interpolation between each new number.

## Syntax

```
ares randomi kmin, kmax, xcps [,imode] [,ifirstval]
```

```
kres randomi kmin, kmax, kcps [,imode] [,ifirstval]
```

## Initialization

*imode* (optional, default=0) -- first interpolation cycle mode (see below)

*ifirstval* (optional, default=0) -- first output value

## Performance

*kmin* -- minimum range limit

*kmax* -- maximum range limit

*kcps*, *xcps* -- rate of random break-point generation

The *randomi* opcode is similar to *randi* but allows the user to set arbitrary minimum and maximum values.

When *imode* = 0 (the default), the *kmin* argument value is outputted during  $1/kcps$  (resp.  $1/xcps$ ) seconds at the beginning of the note, before the first random number is generated. Then the normal interpolation process takes place, first between *kmin* and the first random number generated, and then between successive generated random numbers, each interpolation cycle having a duration of  $1/kcps$  (resp.  $1/xcps$ ) seconds.

When *imode* = 1, a random number is generated at initialization and interpolation begins immediately between the *kmin* argument value and that random number.

When *imode* = 2, a random number is generated at initialization and interpolation begins immediately between the *ifirstval* argument value and that random number.

When *imode* = 3, two random numbers are generated at initialization as breakpoints for the first interpolation cycle.

## Examples

Here is an example of the *randomi* opcode. It uses the file *randomi.csd* [examples/randomi.csd].

### Example 805. Example of the *randomi* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o randomi.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

    seed 0

; Instrument #1.
instr 1
; Choose a random frequency between 220 and 440.
; Generate new random numbers at 10 Hz.
kmin   init 220
kmax   init 440
kcps   init 10
imode  =   p4
ifstval = p5

    printf_i "\nMode: %d\n", 1, imode
    k1 randomi kmin, kmax, kcps, imode, ifstval
    printks "k1 = %f\n", 0.1, k1
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
; each time with a different mode.
i 1 0 1
i 1 1 1 1
i 1 2 1 2 330
i 1 3 1 3
e

</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like this:

```

Mode: 0
k1 = 220.000000
k1 = 220.000000
k1 = 220.146093
k1 = 246.827703
k1 = 395.595775
...

Mode: 1
k1 = 220.000000
k1 = 224.325329
k1 = 274.370074
k1 = 343.216049
k1 = 414.324347
...

```

```
Mode: 2
k1 = 330.000000
k1 = 292.628171
k1 = 334.519777
k1 = 290.610602
k1 = 394.905366
...
```

```
Mode: 3
k1 = 360.727674
k1 = 431.680412
k1 = 380.625254
k1 = 289.267139
k1 = 303.038109
...
```

## See Also

*randi*, *random*, *randomh*

## Credits

Author: Gabriel Maldonado

Arguments *imode* and *ifirstval* added by François Pinot, Jan. 2011, after a discussion with Peiman Khosravi on the csnd list.

Example written by Kevin Conder, adapted for new args by François Pinot.

# rbjeq

rbjeq — Parametric equalizer and filter opcode with 7 filter types, based on algorithm by Robert Bristow-Johnson.

## Description

Parametric equalizer and filter opcode with 7 filter types, based on algorithm by Robert Bristow-Johnson.

## Syntax

```
ar rbjeq asig, kfco, klvl, kQ, kS[, imode]
```

## Initialization

*imode* ( optional, defaults to zero) - sum of:

- 1: skip initialization (should be used in tied, or re-initialized notes only)

and exactly one of the following values to select filter type:

- 0: resonant lowpass filter. *kQ* controls the resonance: at the cutoff frequency (*kfco*), the amplitude gain is *kQ* (e.g. 20 dB for *kQ* = 10), and higher *kQ* values result in a narrower resonance peak. If *kQ* is set to  $\sqrt{0.5}$  (about 0.7071), there is no resonance, and the filter has a response that is very similar to that of `butterlp`. If *kQ* is less than  $\sqrt{0.5}$ , there is no resonance, and the filter has a -6 dB / octave response from about  $kfco * kQ$  to *kfco*. Above *kfco*, there is always a -12 dB / octave cutoff.



### NOTE

The `rbjeq` lowpass filter is basically the same as `ar pareq asig, kfco, 0, kQ, 2` but is faster to calculate.

- 2: resonant highpass filter. The parameters are the same as for the lowpass filter, but the equivalent filter is `butterhp` if *kQ* is 0.7071, and "ar pareq asig, kfco, 0, kQ, 1" in other cases.
- 4: bandpass filter. *kQ* controls the bandwidth, which is  $kfco / kQ$ , and must be always less than  $sr / 2$ . The bandwidth is measured between -3 dB points (i.e. amplitude gain = 0.7071), beyond which there is a +/- 6 dB / octave slope. This filter type is very similar to `ar butterbp asig, kfco, kfco / kQ`.
- 6: band-reject filter, with the same parameters as the bandpass filter, and a response similar to that of `butterbr`.
- 8: peaking EQ. It has an amplitude gain of 1 (0 dB) at 0 Hz and  $sr / 2$ , and *klvl* at the center frequency (*kfco*). Thus, *klvl* controls the amount of boost (if it is greater than 1), or cut (if it is less than 1). Setting *klvl* to 1 results in a flat response. Similarly to the bandpass and band-reject filters, the bandwidth is determined by  $kfco / kQ$  (which must be less than  $sr / 2$  again); however, this time it is between  $\sqrt{klvl}$  points (or, in other words, half the boost or cut in decibels). NOTE: excessively low or high values of *klvl* should be avoided (especially with 32-bit floats), though the opcode was tested with *klvl* = 0.01 and *klvl* = 100. *klvl* = 0 is always an error, unlike in the case of `pareq`, which does allow a zero level.
- 10: low shelf EQ, controlled by *klvl* and *kS* (*kQ* is ignored by this filter type). There is an amplitude gain of *klvl* at zero frequency, while the level of high frequencies (around  $sr / 2$ ) is not changed. At

the corner frequency (*kfco*), the gain is  $\sqrt{\text{klvl}}$  (half the boost or cut in decibels). The *kS* parameter controls the steepness of the slope of the frequency response (see below).

- 12: high shelf EQ. Very similar to the low shelf EQ, but affects the high frequency range.

The default value for *imode* is zero (lowpass filter, initialization not skipped).

## Performance

*ar* -- the output signal.

*asig* -- the input signal



### NOTE

If the input contains silent sections, on Intel CPUs a significant slowdown can occur due to denormals. In such cases, it is recommended to process the input signal with "denorm" opcode before filtering it with *rbjeq* (and actually many other filters).

*kfco* -- cutoff, corner, or center frequency, depending on filter type, in Hz. It must be greater than zero, and less than  $\text{sr} / 2$  (the range of about  $\text{sr} * 0.0002$  to  $\text{sr} * 0.49$  should be safe).

*klvl* -- level (amount of boost or cut), as amplitude gain (e.g. 1: flat response, 4: 12 dB boost, 0.1: 20 dB cut); zero or negative values are not allowed. It is recognized by the peaking and shelving EQ types (8, 10, 12) only, and is ignored by other filters.

*kQ* -- resonance (also *kfco* / bandwidth in many filter types). Not used by the shelving EQs (*imode* = 10 and 12). The exact meaning of this parameter depends on the filter type (see above), but it should be always greater than zero, and usually (*kfco* / *kQ*) less than  $\text{sr} / 2$ .

*kS* -- shelf slope parameter for shelving filters. Must be greater than zero; a higher value means a steeper slope, with resonance if  $kS > 1$  (however, a too high *kS* value may make the filter unstable). If *kS* is set to exactly 1, the shelf slope is as steep as possible without a resonance. Note that the effect of *kS* - especially if it is greater than 1 - also depends on *klvl*, and it does not have any well defined unit.

## Examples

Here is an example of the *rbjeq* opcode. It uses the file *rbjeq.csd* [examples/rbjeq.csd].

### Example 806. An example of the *rbjeq* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o rbjeq.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
```

```
nchnls = 2
odbfs  = 1

instr 1

imode = p4
a1    vco2    .3, 155.6 ; sawtooth wave
kfco  expon  8000, p3, 200 ; filter frequency
asig  rbjeq   a1, kfco, 1, kfco * 0.005, 1, imode
outs  asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 5 0 ;lowpass
i 1 6 5 2 ;highpass
i 1 12 5 4 ;bandpass
i 1 18 5 8 ;equalizer

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Original algorithm by Robert Bristow-Johnson

Csound orchestra version by Josep M Comajuncosas, Aug 1999

Converted to C (with optimizations and bug fixes) by Istvan Varga, Dec 2002

# readclock

readclock — Reads the value of an internal clock.

## Description

Reads the value of an internal clock.

## Syntax

```
ir readclock inum
```

## Initialization

*inum* -- the number of a clock. There are 32 clocks numbered 0 through 31. All other values are mapped to clock number 32.

*ir* -- value at i-time, of the clock specified by *inum*

## Performance

Between a *clockon* and a *clockoff* opcode, the CPU time used is accumulated in the clock. The precision is machine dependent but is the millisecond range on UNIX and Windows systems. The *readclock* opcode reads the current value of a clock at initialization time.

## Examples

Here is an example of the *readclock* opcode. It uses the file *readclock.csd* [examples/readclock.csd].

### Example 807. Example of the *readclock* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o readclock.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1.
instr 1
; Start clock #1.
clockon 1
```

```
; Do something that keeps Csound busy.
a1 oscili 10000, 440, 1
out a1
; Stop clock #1.
clockoff 1
; Print the time accumulated in clock #1.
i1 readclock 1
print i1
endin

</CsInstruments>
<CsScore>

; Initialize the function tables.
; Table 1: an ordinary sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for one second starting at 0:00.
i 1 0 1
; Play Instrument #1 for one second starting at 0:01.
i 1 1 1
; Play Instrument #1 for one second starting at 0:02.
i 1 2 1
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
instr 1: i1 = 0.000
instr 1: i1 = 90.000
instr 1: i1 = 180.000
```

## See Also

*clockoff, clockon*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
July, 1999

Example written by Kevin Conder.

New in Csound version 3.56

# readf

readf — Read a line of text from an external file.

## Description

Read a line of text from an external file once each k-cycle.

## Syntax

```
Sres, kline readf ifilename
```

## Initialization

*ifilename* -- an integer N denoting a file named "input.N" or a character string (in double quotes, spaces permitted) denoting the external file name. For a string, it may either be a full path name with directory specified or a simple filename. In the later case, the file is sought first in the current directory, then in SSDIR, and finally in SFDIR.

## Performance

*Sres* -- output of the line read from *ifilename*.

*kline* -- line number, or -1 when end of file has been reached.

This opcode allows a line of text to be read from a named external file. There may be any number of *readf* opcodes in an instrument or orchestra but they read separately from the same or different files.

## Examples

Here is an example of the *readf* opcode. It uses the file *readf.csd* [examples/readf.csd].

### Example 808. Example of the *readf* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-n
</CsOptions>

<CsInstruments>
instr 1
  Swd          pwd
               printf_i  "Working directory is '%s'\n", 1, Swd
               prints    "Reading myself =):\n"
read:
  Sline, kLinNum readf "readf.csd"
               printf  "Line %d: %s", kLinNum, kLinNum, Sline
  if kLinNum != -1 then
    kgoto read
  else
```

```
                                turnoff
endif
endin
</CsInstruments>

<CsScore>
i 0 1
e
</CsScore>

</CsoundSynthesizer>
```

## See Also

*readfi*.

## Credits

John ffitch and Joachim Heintz

2012; new in 5.17.12

# readfi

readfi — Read a line of text from an external file.

## Description

Read a line of text from an external file once on initialisation.

## Syntax

```
Sres, iline readfi ifilename
```

## Initialization

*ifilename* -- an integer N denoting a file named "input.N" or a character string (in double quotes, spaces permitted) denoting the external file name. For a string, it may either be a full path name with directory specified or a simple filename. In the later case, the file is sought first in the current directory, then in SSDIR, and finally in SFDIR.

*iline* -- line number, or -1 when end of file has been reached.

*Sres* -- output of the line read from *ifilename*.

This opcode allows a line of text to be read from a named external file. There may be any number of *readfi* opcodes in an instrument or orchestra but they read separately from the same or different files.

## Examples

Here is an example of the *readfi* opcode. It uses the file *readfi.csd* [examples/readfi.csd].

### Example 809. Example of the *readfi* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-n
</CsOptions>

<CsInstruments>
instr 1
  Swd pwd
  printf_i "Working directory is '%s'\n", 1, Swd
  prints "Reading myself =):\n"
read:
  Sline, iLineNumber readfi "readfi.csd"
  printf_i "Line %d: %s", iLineNumber, iLineNumber, Sline
  if iLineNumber != -1 igoto read
endin
</CsInstruments>

<CsScore>
i1 0 0.1
```

```
e  
</CsScore>  
  
</CsoundSynthesizer>
```

## See Also

*readf.*

## Credits

John ffitich and Joachim Heintz

2012; new in 5.17.12

# readk

`readk` — Periodically reads an orchestra control-signal value from an external file.

## Description

Periodically reads an orchestra control-signal value from a named external file in a specific format.

## Syntax

```
kres readk ifilename, iformat, iprd
```

## Initialization

*ifilename* -- an integer N denoting a file named "readk.N" or a character string (in double quotes, spaces permitted) denoting the external file name. For a string, it may either be a full path name with directory specified or a simple filename. In the later case, the file is sought first in the current directory, then in SSDIR, and finally in SFDIR.

*iformat* -- specifies the input data format:

- 1 = 8-bit signed integers (char)
- 4 = 16-bit short integers
- 5 = 32-bit long integers
- 6 = 32-bit floats
- 7 = ASCII long integers (plain text)
- 8 = ASCII floats (plain text)

Note that A-law and U-law formats are not available, and that all formats except the last two are binary. The input file should be a "raw", headerless data file.

*iprd* -- the rate (period) in seconds, rounded to the nearest orchestra control period, at which the signal is read from the input file. A value of 0 implies one control period (the enforced minimum), which will read new values at the orchestra control rate. Longer periods will cause the same values to repeat for more than one control period.

## Performance

*kres* -- output of the signal read from *ifilename*.

This opcode allows a generated control signal value to be read from a named external file. The file should contain no header information but it should contain a regularly sampled time series of control values. For ASCII text formats, the values are assumed to be separated by at least one whitespace character. There may be any number of *readk* opcodes in an instrument or orchestra and they may read from the same or different files.

## Examples

Here is an example of the *readk* opcode. It uses the file *readk.csd* [examples/readk.csd].

**Example 810. Example of the readk opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o readk.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

0dbfs = 1
; By Andres Cabrera 2008

instr 1
; Read a number from the file every 0.5 seconds
kfibo readk "fibonacci.txt", 7, 0.5
kpitchclass = 8 + ((kfibo % 12)/100)
printk2 kpitchclass
kcps = cpspch( kpitchclass )
printk2 kcps
a1 oscil 0.5, kcps, 1
out a1
endin

</CsInstruments>
<CsScore>
f 1 0 1024 10 1
i 1 0 10
e

</CsScore>
</CsoundSynthesizer>

```

Here is another example of the readk opcode. It uses the file *readk-2.csd* [examples/readk-2.csd].

**Example 811. Example 2 of the readk opcode.**

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc      ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o readk-2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

```

```
Odbfs = 1

giSine ftgen 0, 0, 2^10, 10, 1

instr 1 ;writes a control signal to a file
kfreq randh 100, 1, 2, 1, 500 ;generates one random number between 400 and 600 per second
dumpk kfreq, "dumpk.txt", 8, 1 ;writes the control signal
printk 1, kfreq ;prints it
endin

instr 2 ;reads the file written by instr 1
kfreq readk "dumpk.txt", 8, 1
printk 1, kfreq ;prints it
aout poscil .2, kfreq, giSine
outs aout, aout
endin

</CsInstruments>
<CsScore>
i 1 0 5
i 2 5 5
e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
WARNING: Seeding from current time 683384022
```

```
i 1 time 1.00033: 463.64510
i 1 time 2.00000: 463.64510
i 1 time 3.00000: 483.14200
i 1 time 4.00000: 567.55973
i 1 time 5.00000: 576.37060
i 1 time 6.00000: 460.66550

i 2 time 6.00033: 463.64510
i 2 time 7.00000: 463.64510
i 2 time 8.00000: 483.14200
i 2 time 9.00000: 567.55970
i 2 time 10.00000: 576.37060
i 2 time 11.00000: 460.66550
```

## See Also

*dumpk, dumpk2, dumpk3, dumpk4, readk2, readk3, readk4*

## Credits

By: John ffitich and Barry L. Vercoe

1999 or earlier

# readk2

`readk2` — Periodically reads two orchestra control-signal values from an external file.

## Description

Periodically reads two orchestra control-signal values from an external file.

## Syntax

```
kr1, kr2 readk2 ifilename, iformat, iprd
```

## Initialization

*ifilename* -- an integer N denoting a file named "readk.N" or a character string (in double quotes, spaces permitted) denoting the external file name. For a string, it may either be a full path name with directory specified or a simple filename. In the later case, the file is sought first in the current directory, then in *SSDIR*, and finally in *SFDIR*.

*iformat* -- specifies the input data format:

- 1 = 8-bit signed integers (char)
- 4 = 16-bit short integers
- 5 = 32-bit long integers
- 6 = 32-bit floats
- 7 = ASCII long integers (plain text)
- 8 = ASCII floats (plain text)

Note that A-law and U-law formats are not available, and that all formats except the last two are binary. The input file should be a "raw", headerless data file.

*iprd* -- the rate (period) in seconds, rounded to the nearest orchestra control period, at which the signals are read from the input file. A value of 0 implies one control period (the enforced minimum), which will read new values at the orchestra control rate. Longer periods will cause the same values to repeat for more than one control period.

## Performance

*kr1*, *kr2* -- output of the signals read from *ifilename*.

This opcode allows two generated control signal values to be read from a named external file. The file should contain no header information but it should contain a regularly sampled time series of control values. For binary formats, the individual samples of each signal are interleaved. For ASCII text formats, the values are assumed to be separated by at least one whitespace character. The two "channels" in a sample frame may be on the same line or separated by newline characters, it does not matter. There may be any number of *readk2* opcodes in an instrument or orchestra and they may read from the same or different files.

## Examples

Here is an example of the readk2 opcode. It uses the file *readk2.csd* [examples/readk2.csd].

### Example 812. Example of the readk2 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o readk2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine ftgen 0, 0, 2^10, 10, 1

instr 1 ;writes two control signals to a file
kfreq  randh 100, 1, 2, 1, 500 ;generates one random number between 400 and 600 per second
kdb    randh 12, 1, 2, 1, -12 ;amplitudes in dB between -24 and 0
      dumpk2 kfreq, kdb, "dumpk2.txt", 8, 1 ;writes the control signals
      prints "WRITING:\n"
      printks "kfreq = %f, kdb = %f\n", 1, kfreq, kdb ;prints them
endin

instr 2 ;reads the file written by instr 1
kf,kdb readk2 "dumpk2.txt", 8, 1
      prints "READING:\n"
      printks "kfreq = %f, kdb = %f\n", 1, kf, kdb ;prints again
kdb    lineto kdb, .1 ;smoothing amp transition
aout   poscil ampdb(kdb), kf, giSine
      outs  aout, aout
endin

</CsInstruments>
<CsScore>
i 1 0 5
i 2 5 5
e
</CsScore>
</CsoundSynthesizer>

```

The output should include lines like these:

```

kfreq = 429.202551, kdb = -20.495694
kfreq = 429.202551, kdb = -20.495694
kfreq = 407.275258, kdb = -23.123776
kfreq = 475.264472, kdb = -9.300846
kfreq = 569.979181, kdb = -7.315527
kfreq = 440.103457, kdb = -0.058331

kfreq = 429.202600, kdb = -20.495700
kfreq = 429.202600, kdb = -20.495700
kfreq = 407.275300, kdb = -23.123800

```

kfreq = 475.264500, kdb = -9.300800  
kfreq = 569.979200, kdb = -7.315500  
kfreq = 440.103500, kdb = -0.058300

## See Also

*dumpk, dumpk2, dumpk3, dumpk4, readk, readk3, readk4*

## Credits

By: John ffitich and Barry L. Vercoe

1999 or earlier

# readk3

readk3 — Periodically reads three orchestra control-signal values from an external file.

## Description

Periodically reads three orchestra control-signal values from an external file.

## Syntax

```
kr1, kr2, kr3 readk3 ifilename, iformat, iprd
```

## Initialization

*ifilename* -- an integer N denoting a file named "readk.N" or a character string (in double quotes, spaces permitted) denoting the external file name. For a string, it may either be a full path name with directory specified or a simple filename. In the later case, the file is sought first in the current directory, then in *SSDIR*, and finally in *SFDIR*.

*iformat* -- specifies the input data format:

- 1 = 8-bit signed integers (char)
- 4 = 16-bit short integers
- 5 = 32-bit long integers
- 6 = 32-bit floats
- 7 = ASCII long integers (plain text)
- 8 = ASCII floats (plain text)

Note that A-law and U-law formats are not available, and that all formats except the last two are binary. The input file should be a "raw", headerless data file.

*iprd* -- the rate (period) in seconds, rounded to the nearest orchestra control period, at which the signals are read from the input file. A value of 0 implies one control period (the enforced minimum), which will read new values at the orchestra control rate. Longer periods will cause the same values to repeat for more than one control period.

## Performance

*kr1*, *kr2*, *kr3* -- output of the signals read from *ifilename*.

This opcode allows three generated control signal values to be read from a named external file. The file should contain no header information but it should contain a regularly sampled time series of control values. For binary formats, the individual samples of each signal are interleaved. For ASCII text formats, the values are assumed to be separated by at least one whitespace character. The three "channels" in a sample frame may be on the same line or separated by newline characters, it does not matter. There may be any number of *readk3* opcodes in an instrument or orchestra and they may read from the same or different files.

## Examples

Here is an example of the readk3 opcode. It uses the file *readk3.csd* [examples/readk3.csd].

### Example 813. Example of the readk3 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o readk3.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine ftgen 0, 0, 2^10, 10, 1

instr 1 ;writes three control signals to a file
kfreq  randh    100, 1, 2, 1, 500 ;generates one random number between 400 and 600 per second
kdb    randh    12, 1, 2, 1, -12 ;amplitudes in dB between -24 and 0
kpan   randh    .5, 1, 2, 1, .5 ;panning between 0 and 1
      dumpk3    kfreq, kdb, kpan, "dumpk3.txt", 8, 1 ;writes the control signals
      prints    "WRITING:\n"
      printks   "kfreq = %f, kdb = %f, kpan = %f\n", 1, kfreq, kdb, kpan ;prints them
endin

instr 2 ;reads the file written by instr 1
kf,kdb,kp readk3 "dumpk3.txt", 8, 1
      prints   "READING:\n"
      printks  "kfreq = %f, kdb = %f, kpan = %f\n", 1, kf, kdb, kp ;prints again
kdb    lineto  kdb, .1 ;smoothing amp transition
kp     lineto  kp, .1 ;smoothing pan transition
aout   poscil  ampdb(kdb), kf, giSine
aL, aR pan2    aout, kp
      outs    aL, aR
endin

</CsInstruments>
<CsScore>
i 1 0 5
i 2 5 5
e
</CsScore>
</CsoundSynthesizer>

```

The output should include lines like these:

```

WRITING:
kfreq = 473.352855, kdb = -15.197657, kpan = 0.366764
kfreq = 473.352855, kdb = -15.197657, kpan = 0.366764
kfreq = 441.426368, kdb = -19.026206, kpan = 0.207327
kfreq = 452.965140, kdb = -21.447486, kpan = 0.553270
kfreq = 585.106328, kdb = -11.903852, kpan = 0.815665
kfreq = 482.056760, kdb = -4.046744, kpan = 0.876537

```

READING:

kfreq = 473.352900, kdb = -15.197700, kpan = 0.366800  
kfreq = 473.352900, kdb = -15.197700, kpan = 0.366800  
kfreq = 441.426400, kdb = -19.026200, kpan = 0.207300  
kfreq = 452.965100, kdb = -21.447500, kpan = 0.553300  
kfreq = 585.106300, kdb = -11.903900, kpan = 0.815700  
kfreq = 482.056800, kdb = -4.046700, kpan = 0.876500

## See Also

*dumpk, dumpk2, dumpk3, dumpk4, readk, readk2, readk4*

## Credits

By: John ffitc and Barry L. Vercoe

1999 or earlier

# readk4

readk4 — Periodically reads four orchestra control-signal values from an external file.

## Description

Periodically reads four orchestra control-signal values from an external file.

## Syntax

```
kr1, kr2, kr3, kr4 readk4 ifilename, iformat, iprd
```

## Initialization

*ifilename* -- an integer N denoting a file named "readk.N" or a character string (in double quotes, spaces permitted) denoting the external file name. For a string, it may either be a full path name with directory specified or a simple filename. In the later case, the file is sought first in the current directory, then in *SSDIR*, and finally in *SFDIR*.

*iformat* -- specifies the input data format:

- 1 = 8-bit signed integers (char)
- 4 = 16-bit short integers
- 5 = 32-bit long integers
- 6 = 32-bit floats
- 7 = ASCII long integers (plain text)
- 8 = ASCII floats (plain text)

Note that A-law and U-law formats are not available, and that all formats except the last two are binary. The input file should be a "raw", headerless data file.

*iprd* -- the rate (period) in seconds, rounded to the nearest orchestra control period, at which the signals are read from the input file. A value of 0 implies one control period (the enforced minimum), which will read new values at the orchestra control rate. Longer periods will cause the same values to repeat for more than one control period.

## Performance

*kr1, kr2, kr3, kr4* -- output of the signals read from *ifilename*.

This opcode allows four generated control signal values to be read from a named external file. The file should contain no header information but it should contain a regularly sampled time series of control values. For binary formats, the individual samples of each signal are interleaved. For ASCII text formats, the values are assumed to be separated by at least one whitespace character. The four "channels" in a sample frame may be on the same line or separated by newline characters, it does not matter. There may be any number of *readk4* opcodes in an instrument or orchestra and they may read from the same or different files.

## Examples

Here is an example of the `readk4` opcode. It uses the file `readk4.csd` [examples/readk4.csd].

### Example 814. Example of the `readk4` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o readk4.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine ftgen 0, 0, 2^10, 10, 1

instr 1 ;writes four control signals to a file
kcf      randh    950, 1, 2, 1, 1050 ;generates one random number between 100 and 2000 per second
kq       randh    10, 1, 2, 1, 11 ;generates another random number between 1 and 21 per second
kdb      randh    9, 1, 2, 1, -15 ;amplitudes in dB between -24 and -6
kpan     randh    .5, 1, 2, 1, .5 ;panning between 0 and 1
          dumpk4  kcf, kq, kdb, kpan, "dumpk4.txt", 8, 1 ;writes the control signals
          prints  "WRITING:\n"
          printks "%kcf = %f, kq = %f, kdb = %f, kpan = %f\n", 1, kcf, kq, kdb, kpan ;prints them
endin

instr 2 ;reads the file written by instr 1
kcf,kq,kdb,kp readk4 "dumpk4.txt", 8, 1
          prints  "READING:\n"
          printks "%kcf = %f, kq = %f, kdb = %f, kpan = %f\n", 1, kcf, kq, kdb, kp ;prints values
kdb      lineto   kdb, .1 ;smoothing amp transition
kp       lineto   kp, .1 ;smoothing pan transition
anoise   rand     ampdb(kdb), 2, 1
kbw      =        kcf/kq ;bandwidth of resonant filter
abp      reson    anoise, kcf, kbw
aout     balance  abp, anoise
aL, aR   pan2     aout, kp
          outs     aL, aR
endin

</CsInstruments>
<CsScore>
i 1 0 5
i 2 5 5
e
</CsScore>
</CsoundSynthesizer>
```

The output should include lines like these:

```
WRITING:
kcf = 1122.469723, kq = 11.762839, kdb = -14.313445, kpan = 0.538142
kcf = 1122.469723, kq = 11.762839, kdb = -14.313445, kpan = 0.538142
```

```
kcf = 1148.638412, kq = 12.040490, kdb = -14.061868, kpan = 0.552205  
kcf = 165.796855, kq = 18.523179, kdb = -15.816977, kpan = 0.901528  
kcf = 147.729960, kq = 13.071911, kdb = -11.924531, kpan = 0.982518  
kcf = 497.430113, kq = 13.605512, kdb = -21.586611, kpan = 0.179229
```

READING:

WARNING: Seeding from current time 3308160476

```
kcf = 1122.469700, kq = 11.762800, kdb = -14.313400, kpan = 0.538100  
kcf = 1122.469700, kq = 11.762800, kdb = -14.313400, kpan = 0.538100  
kcf = 1148.638400, kq = 12.040500, kdb = -14.061900, kpan = 0.552200  
kcf = 165.796900, kq = 18.523200, kdb = -15.817000, kpan = 0.901500  
kcf = 147.730000, kq = 13.071900, kdb = -11.924500, kpan = 0.982500  
kcf = 497.430100, kq = 13.605500, kdb = -21.586600, kpan = 0.179200
```

## See Also

*dumpk, dumpk2, dumpk3, dumpk4, readk, readk2, readk3*

## Credits

By: John ffitc and Barry L. Vercoe

1999 or earlier

# readscore

readscore — Read, preprocess and schedule a score from an input string.

## Description

Readscore will issue one or more score events. It can handle strings in the same conditions as the standard score, including preprocessing (carry, sort, ramp, etc). Multi-line strings are accepted, using `{{ }}` to enclose the string.

## Syntax

```
readscore Sin
```

## Initialization

“*Sin*” -- a string (in double-quotes or enclosed by `{{ }}`) containing one or more score events.

## Examples

Here is an example of the readscore opcode. It uses the file `readscore.csd` [examples/readscore.csd].

### Example 815. Example of the readscore opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-d -o dac
</CsOptions>
<CsInstruments>

instr 1
a1 flooper2 1000,p4,2,5,0.1,1
  out a1
endin

instr 2
ires readscore {{
{12 COUNT
i1 $COUNT 1 [1 + $COUNT/12]
}}
endin

</CsInstruments>
<CsScore>
f0 12
f 1 0 0 1 "fox.wav" 0 0 1

i2 0 1

</CsScore>
```

```
</CsoundSynthesizer>
```

You can use string opcodes like *sprintfk* to produce strings to be passed to *readscore* like this:

```
Sfil = "/Volumes/Bla/file.aif"  
String sprintfk {{i 2 0 %f "%s" %f %f %f %f}}, idur, Sfil, p5, p6, knorm, iskip  
readscore String
```

## See also

*event, event\_i, schedule, schedwhen, schedkwhen, schedkwhennamed*

## Credits

Author: Victor Lazzarini, 2013

# readscratch

readscratch — returns a value stored in the instance of an instrument.

## Description

The *readscratch* opcode returns one of four scalar values stored in the instance of an instrument.

## Syntax

```
ival readscratch[index]
```

## Initialisation

*ival* -- variable for result.

*index* -- which value to read, defaulting to zero.

## Examples

Here is an example of the readscratch opcode. It uses the file *readscratch.csd* [examples/readscratch.csd].

### Example 816. Example of the readscratch opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ilast readscratch
writescratch p2
ilen readscratch 1
writescratch p3, 1
printf_i "last run at %f for %f\n", ilen, ilast, ilen
endin

</CsInstruments>
<CsScore>
i 1 0 1
i 1 2 3
i 1 6 10
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*writescratch*,

## Credits

Author: John ffitch  
March 2013

New in Csound version 6.00

# rect2pol

rect2pol — Rectangular to polar format conversion.

## Description

Converts an input array in real-imaginary format to magnitude-phase format.

## Syntax

```
kout[] rect2pol kin[]
```

## Performance

*kout[]* -- output array containing the complex-valued magnitude-phase output. It will be created if it does not exist.

*kin[]* -- input array containing the complex-valued real-imaginary input.

## Examples

Here is an example of the rect2pol opcode. It uses the file *rect2pol.csd* [examples/rect2pol.csd].

### Example 817. Example of the rect2pol opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-d -o dac
</CsOptions>
<CsInstruments>
/* ksmpls needs to be an integer div of
   hopsize */
ksmps = 64

instr 1

  ihopsize = 256 ; hopsize
  ifftsize = 1024 ; FFT size
  iolaps = ifftsize/ihopsize ; overlaps
  ibw = sr/ifftsize ; bin bandwidth
  kcnt init 0 ; counting vars
  krow init 0

  kOla[] init ifftsize ; overlap-add buffer
  kIn[] init ifftsize ; input buffer
  kOut[][] init iolaps, ifftsize ; output buffers

  a1 diskin2 "fox.wav",1,0,1 ; audio input

  /* every hopsize samples */
  if kcnt == ihopsize then
    /* window and take FFT */
    kWin[] window kIn,krow*ihopsize
```

```
kSpec[] rfft kWin

kSpec rect2pol kSpec

/* reduce mags between high and low freqs */
ilow = 0
ihigh = 1000
ki = int(ilow/ibw)
until ki >= int(ihigh/ibw) do
  kSpec[ki] = kSpec[ki]*0.1
  ki += 2
od

kSpec pol2rect kSpec

/* IFFT + window */
kRow[] rfft kSpec
kWin window kRow, krow*ihopsize
/* place it on out buffer */
kOut setrow kWin, krow

/* zero the ola buffer */
kOla = 0
/* overlap-add */
ki = 0
until ki == iolaps do
  kRow getrow kOut, ki
  kOla = kOla + kRow
  ki += 1
od

/* update counters */
krow = (krow+1)%iolaps
kcnt = 0
endif

/* shift audio in/out of buffers */
kIn shiftin a1
a2 shiftout kOla
out a2/iolaps

/* increment counter */
kcnt += ksmps

endin

</CsInstruments>
<CsScore>
i1 0 10
</CsScore>
</CsoundSynthesizer>
```

## See Also

Vectorial opcodes, array opcodes

## Credits

Author: Victor Lazzarini  
NUI Maynooth  
2014

New in version 6.04

# reinit

reinit — Suspends a performance while a special initialization pass is executed.

## Description

Suspends a performance while a special initialization pass is executed.

Whenever this statement is encountered during a p-time pass, performance is temporarily suspended while a special Initialization pass, beginning at *label* and continuing to *rireturn* or *endin*, is executed. Performance will then be resumed from where it left off.

## Syntax

```
reinit label
```

## Examples

The following statements will generate an exponential control signal whose value moves from 440 to 880 exactly ten times over the duration p3. They use the file *reinit.csd* [examples/reinit.csd].

### Example 818. Example of the reinit opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o reinit.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

instr 1

reset:
  timeout 0, p3/10, contin
  reinit reset

contin:
  kLine expon 440, p3/10, 880
  aSig oscil 10000, kLine, 1
  out aSig
  rireturn

endin

</CsInstruments>
```

```
<CsScore>
f1 0 4096 10 1
i1 0 10
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*rigoto, rireturn*

# release

`release` — Indicates whether a note is in its “release” stage.

## Description

Provides a way of knowing when a note off message for the current note is received. Only a noteoff message with the same MIDI note number as the one which triggered the note will be reported by *release*.

## Syntax

```
kflag release
```

## Performance

*kflag* -- indicates whether the note is in its “release” stage. (1 if a note off is received, otherwise 0)

*release* outputs current note state. If current note is in the “release” stage (i.e. if its duration has been extended with *xtratim* opcode and if it has only just deactivated), then the *kflag* output argument is set to 1. Otherwise (in sustain stage of current note), *kflag* is set to 0.

This opcode is useful for implementing complex release-oriented envelopes. When used in conjunction with *xtratim* it can provide an alternative to the hard-coded behaviour of the “r” opcodes (*linsegr*, *expsegr* et al), where release time is set to the longest time specified in the active instrument.

## Examples

See the examples for *xtratim*.

## See Also

*xtratim*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

# remoteport

remoteport — Defines the port for use with the remote system.

## Description

Defines the port for use with the *insremot*, *midremot*, *insglobal* and *midglobal* opcodes.

## Syntax

```
remoteport iportnum
```

## Initialization

*iportnum* -- number of the port to be used. If zero or negative the default port 40002 is selected.

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
November, 2006

New in Csound version 5.05

## remove

`remove` — Removes the definition of an instrument.

## Description

Removes the definition of an instrument as long as it is not in use.

## Syntax

```
remove insnum
```

## Initialization

*insnum* -- number or name of the instrument to be deleted

## Performance

As long as the indicated instrument is not active, *remove* deletes the instrument and memory associated with it. It should be treated with care as it is possible that in some cases its use may lead to a crash.

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
June, 2006

New in Csound version 5.04

# repluck

repluck — Physical model of the plucked string.

## Description

*repluck* is an implementation of the physical model of the plucked string. A user can control the pluck point, the pickup point, the filter, and an additional audio signal, *axcite*. *axcite* is used to excite the 'string'. Based on the Karplus-Strong algorithm.

## Syntax

```
ares repluck iplk, kamp, icps, kpick, krefl, axcite
```

## Initialization

*iplk* -- The point of pluck is *iplk*, which is a fraction of the way up the string (0 to 1). A pluck point of zero means no initial pluck.

*icps* -- The string plays at *icps* pitch.

## Performance

*kamp* -- Amplitude of note.

*kpick* -- Proportion of the way along the string to sample the output.

*krefl* -- the coefficient of reflection, indicating the lossiness and the rate of decay. It must be strictly between 0 and 1 (it will complain about both 0 and 1).

## Performance

*axcite* -- A signal which excites the string.

## Examples

Here is an example of the repluck opcode. It uses the file *repluck.csd* [examples/repluck.csd].

### Example 819. Example of the repluck opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o repluck.wav -W ;; for file output any platform
</CsOptions>
```

```
<CsInstruments>

sr = 44100
ksmps = 32
Odbfs = 1
nchnls = 2

instr 1

iplk = 0.75
kamp = .8
icps = 110
krefl = p4
kpick = p5

axcite oscil 1, 1, 1
asig repluck iplk, kamp, icps, kpick, krefl, axcite
asig dcblock2 asig ;get rid of DC offset
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine wave.

s
i 1 0 1 0.95 0.75 ;sounds heavier (=p5)
i 1 + 1 <
i 1 + 1 <
i 1 + 1 <
i 1 + 10 0.6

s
i 1 0 1 0.95 0.15 ;sounds softer (=p5)
i 1 + 1 <
i 1 + 1 <
i 1 + 1 <
i 1 + 10 0.6
e
</CsScore>
</CsSoundSynthesizer>
```

## See Also

*wgpluck2*

## Credits

Author: John ffitth  
University of Bath/Codemist Ltd.  
Bath, UK  
1997

New in version 3.47

# reson

reson — A second-order resonant filter.

## Description

A second-order resonant filter.

## Syntax

```
ares reson asig, xcf, xbw [, iscl] [, iskip]
```

## Initialization

*iscl* (optional, default=0) -- coded scaling factor for resonators. A value of 1 signifies a peak response factor of 1, i.e. all frequencies other than *kcf* are attenuated in accordance with the (normalized) response curve. A value of 2 raises the response factor so that its overall RMS value equals 1. (This intended equalization of input and output power assumes all frequencies are physically present; hence it is most applicable to white noise.) A zero value signifies no scaling of the signal, leaving that to some later adjustment (see *balance*). The default value is 0.

*iskip* (optional, default=0) -- initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*ares* -- the output signal at audio rate.

*asig* -- the input signal at audio rate.

*xcf* -- the center frequency of the filter, or frequency position of the peak response.

*xbw* -- bandwidth of the filter (the Hz difference between the upper and lower half-power points).

*reson* is a second-order filter in which *kcf* controls the center frequency, or frequency position of the peak response, and *kbw* controls its bandwidth (the frequency difference between the upper and lower half-power points).

## Examples

Here is an example of the reson opcode. It uses the file *reson.csd* [examples/reson.csd].

### Example 820. Example of the reson opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform
```

```
-odac    ;;;realtime audio out
;-iadc   ;;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o reson.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 2

instr 1

asaw vco2 .2, 220 ;sawtooth
kcf line 220, p3, 1760 ;vary cut-off frequency from 220 to 1280 Hz
kbw = p4 ;vary bandwidth of filter too
ares reson asaw, kcf, kbw
asig balance ares, asaw
outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 4 10 ;bandwidth of filter = 10 Hz
i 1 + 4 50 ;50 Hz and
i 1 + 4 200 ;200 Hz
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*areson, aresonk, atone, atonek, port, portk, resonk, tone, tonek*

## Credits

Audio rate parameters introduced in version 6.02

November 2013.

# resonk

resonk — A second-order resonant filter.

## Description

A second-order resonant filter.

## Syntax

```
kres resonk ksig, kcf, kbw [, iscl] [, iskip]
```

## Initialization

*iscl* (optional, default=0) -- coded scaling factor for resonators. A value of 1 signifies a peak response factor of 1, i.e. all frequencies other than *kcf* are attenuated in accordance with the (normalized) response curve. A value of 2 raises the response factor so that its overall RMS value equals 1. (This intended equalization of input and output power assumes all frequencies are physically present; hence it is most applicable to white noise.) A zero value signifies no scaling of the signal, leaving that to some later adjustment (see *balance*). The default value is 0.

*iskip* (optional, default=0) -- initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*kres* -- the output signal at control-rate.

*ksig* -- the input signal at control-rate.

*kcf* -- the center frequency of the filter, or frequency position of the peak response.

*kbw* -- bandwidth of the filter (the Hz difference between the upper and lower half-power points).

*resonk* is like *reson* except its output is at control-rate rather than audio rate.

## Examples

Here is an example of the resonk opcode. It uses the file *resonk.csd* [examples/resonk.csd].

### Example 821. Example of the resonk opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsoundOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
```

```
; -o resonk.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gisin ftgen 0, 0, 2^10, 10, 1

instr 1

ksig randomh 400, 1800, 150
aout poscil .2, 1000+ksig, gisin
outs aout, aout
endin

instr 2

ksig randomh 400, 1800, 150
khp line 1, p3, 400 ;vary high-pass
ksig resonk ksig, khp, 50
aout poscil .2, 1000+ksig, gisin
outs aout, aout
endin

</CsInstruments>
<CsScore>
i 1 0 5
i 2 5.5 5
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*areson, aresonk, atone, atonek, port, portk, reson, tone, tonek*

## Credits

Author: Robin Whittle  
Australia  
May 1997

# resonr

`resonr` — A bandpass filter with variable frequency response.

## Description

Implementations of a second-order, two-pole two-zero bandpass filter with variable frequency response.

## Syntax

```
ares resonr asig, xcf, xbw [, iscl] [, iskip]
```

## Initialization

The optional initialization variables for `resonr` are identical to the i-time variables for `reson`.

`iscl` (optional, default=0) -- coded scaling factor for resonators. A value of 1 signifies a peak response factor of 1, i.e. all frequencies other than `xcf` are attenuated in accordance with the (normalized) response curve. A value of 2 raises the response factor so that its overall RMS value equals 1. This intended equalization of input and output power assumes all frequencies are physically present; hence it is most applicable to white noise. A zero value signifies no scaling of the signal, leaving that to some later adjustment (see *balance*). The default value is 0.

`iskip` (optional, default=0) -- initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

`asig` -- input signal to be filtered

`xcf` -- cutoff or resonant frequency of the filter, measured in Hz

`xbw` -- bandwidth of the filter (the Hz difference between the upper and lower half-power points)

`resonr` and `resonz` are variations of the classic two-pole bandpass resonator (`reson`). Both filters have two zeroes in their transfer functions, in addition to the two poles. `resonz` has its zeroes located at  $z = 1$  and  $z = -1$ . `resonr` has its zeroes located at  $+\sqrt{R}$  and  $-\sqrt{R}$ , where  $R$  is the radius of the poles in the complex  $z$ -plane. The addition of zeroes to `resonr` and `resonz` results in the improved selectivity of the magnitude response of these filters at cutoff frequencies close to 0, at the expense of less selectivity of frequencies above the cutoff peak.

`resonr` and `resonz` are very close to constant-gain as the center frequency is swept, resulting in a more efficient control of the magnitude response than with traditional two-pole resonators such as `reson`.

`resonr` and `resonz` produce a sound that is considerably different from `reson`, especially for lower center frequencies; trial and error is the best way of determining which resonator is best suited for a particular application.

## Examples

Here is an example of the `resonr` and `resonz` opcodes. It uses the file `resonr.csd` [examples/resonr.csd].

**Example 822. Example of the resonr and resonz opcodes.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o resonr.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

/* Written by Sean Costello */
; Orchestra file for resonant filter sweep of a sawtooth-like waveform.
; The outputs of reson, resonr, and resonz are scaled by coefficients
; specified in the score, so that each filter can be heard on its own
; from the same instrument.

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 1

idur      =      p3
ibegfreq  =      p4                ; beginning of sweep frequency
iendfreq  =      p5                ; ending of sweep frequency
ibw       =      p6                ; bandwidth of filters in Hz
ifreq     =      p7                ; frequency of gbuzz that is to be filtered
iamp      =      p8                ; amplitude to scale output by
ires      =      p9                ; coefficient to scale amount of reson in output
iresr     =      p10               ; coefficient to scale amount of resonr in output
iresz     =      p11               ; coefficient to scale amount of resonz in output

; Frequency envelope for reson cutoff
kfreq     linseg ibegfreq, idur * .5, iendfreq, idur * .5, ibegfreq

; Amplitude envelope to prevent clicking
kenv      linseg 0, .1, iamp, idur - .2, iamp, .1, 0

; Number of harmonics for gbuzz scaled to avoid aliasing
iharms    =      (sr*.4)/ifreq

asig      gbuzz 1, ifreq, iharms, 1, .9, 1      ; "Sawtooth" waveform
ain       =      kenv * asig                    ; output scaled by amp envelope
ares      reson ain, kfreq, ibw, 1
aresr     resonr ain, kfreq, ibw, 1
aresz     resonz ain, kfreq, ibw, 1

          out ares * ires + aresr * iresr + aresz * iresz

endin

</CsInstruments>
<CsScore>

/* Written by Sean Costello */
f1 0 8192 9 1 1 .25                ; cosine table for gbuzz generator

i1 0 10 1 3000 200 100 4000 1 0 0    ; reson  output with bw = 200
i1 10 10 1 3000 200 100 4000 0 1 0   ; resonr output with bw = 200

```

```
i1 20 10 1 3000 200 100 4000 0 0 1 ; resonz output with bw = 200
i1 30 10 1 3000 50 200 8000 1 0 0 ; reson output with bw = 50
i1 40 10 1 3000 50 200 8000 0 1 0 ; resonr output with bw = 50
i1 50 10 1 3000 50 200 8000 0 0 1 ; resonz output with bw = 50
e

</CsScore>
</CsoundSynthesizer>
```

## Technical History

*resonr* and *resonz* were originally described in an article by Julius O. Smith and James B. Angell.<sup>1</sup> Smith and Angell recommended the *resonz* form (zeros at +1 and -1) when computational efficiency was the main concern, as it has one less multiply per sample, while *resonr* (zeroes at + and - the square root of the pole radius R) was recommended for situations when a perfectly constant-gain center peak was required.

Ken Steiglitz, in a later article<sup>2</sup>, demonstrated that *resonz* had constant gain at the true peak of the filter, as opposed to *resonr*, which displayed constant gain at the pole angle. Steiglitz also recommended *resonz* for its sharper notches in the gain curve at zero and Nyquist frequency. Steiglitz's recent book<sup>3</sup> features a thorough technical discussion of *reson* and *resonz*, while Dodge and Jerse's textbook<sup>4</sup> illustrates the differences in the response curves of *reson* and *resonz*.

## References

1. Smith, Julius O. and Angell, James B., "A Constant-Gain Resonator Tuned by a Single Coefficient," *Computer Music Journal*, vol. 6, no. 4, pp. 36-39, Winter 1982.
2. Steiglitz, Ken, "A Note on Constant-Gain Digital Resonators," *Computer Music Journal*, vol. 18, no. 4, pp. 8-10, Winter 1994.
3. Ken Steiglitz, *A Digital Signal Processing Primer, with Applications to Digital Audio and Computer Music*. Addison-Wesley Publishing Company, Menlo Park, CA, 1996.
4. Dodge, Charles and Jerse, Thomas A., *Computer Music: Synthesis, Composition, and Performance*. New York: Schirmer Books, 1997, 2nd edition, pp. 211-214.

## See Also

*resonz*

## Credits

Author: Sean Costello  
Seattle, Washington  
1999

New in Csound version 3.55

Audio rate parameters introduced in version 6.02

November 2013.

# resonx

resonx — Emulates a stack of filters using the reson opcode.

## Description

*resonx* is equivalent to a filters consisting of more layers of *reson* with the same arguments, serially connected. Using a stack of a larger number of filters allows a sharper cutoff. They are faster than using a larger number instances in a Csound orchestra of the old opcodes, because only one initialization and k-cycle are needed at time and the audio loop falls entirely inside the cache memory of processor.

## Syntax

```
ares resonx asig, xcf, xbw [, inumlayer] [, iscl] [, iskip]
```

## Initialization

*inumlayer* (optional) -- number of elements in the filter stack. Default value is 4.

*iscl* (optional, default=0) -- coded scaling factor for resonators. A value of 1 signifies a peak response factor of 1, i.e. all frequencies other than kcf are attenuated in accordance with the (normalized) response curve. A value of 2 raises the response factor so that its overall RMS value equals 1. (This intended equalization of input and output power assumes all frequencies are physically present; hence it is most applicable to white noise.) A zero value signifies no scaling of the signal, leaving that to some later adjustment (see *balance*). The default value is 0.

*iskip* (optional, default=0) -- initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*asig* -- input signal

*xcf* -- the center frequency of the filter, or frequency position of the peak response.

*xbw* -- bandwidth of the filter (the Hz difference between the upper and lower half-power points)

## Examples

Here is an example of the resonx opcodes. It uses the file *resonx.csd* [examples/resonx.csd].

### Example 823. Example of the resonx opcodes.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;;realtime audio out
```

```
;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o resonx.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; unfiltered noise

kenv linseg 0, p3*.5, 1, p3*.5, 0 ;envelope
asig rand 0.7 ;white noise
outs asig*kenv, asig*kenv

endin

instr 2 ; filtered noise

kenv linseg 0, p3*.5, 1, p3*.5, 0 ;envelope
asig rand 0.7
kcf line 300, p3, 2000
afilt resonx asig, kcf, 300, 4
asig balance afilt, asig
outs asig*kenv, asig*kenv

endin
</CsInstruments>
<CsScore>

i 1 0 2
i 2 3 2

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*atonex*, *tonex*

## Credits

Author: Gabriel Maldonado (adapted by John ffitich)  
Italy

New in Csound version 3.49

Audio rate parameters introduced in version 6.02

November 2013.

# resonxk

resonxk — Control signal resonant filter stack.

## Description

*resonxk* is equivalent to a group of resonk filters, with the same arguments, serially connected. Using a stack of a larger number of filters allows a sharper cutoff.

## Syntax

```
kres resonxk ksig, kcf, kbw[, inumlayer, iscl, istor]
```

## Initialization

*inumlayer* - number of elements of filter stack. Default value is 4. Maximum value is 10

*iscl* (optional, default=0) - coded scaling factor for resonators. A value of 1 signifies a peak response factor of 1, i.e. all frequencies other than kcf are attenuated in accordance with the (normalized) response curve. A value of 2 raises the response factor so that its overall RMS value equals 1. (This intended equalization of input and output power assumes all frequencies are physically present; hence it is most applicable to white noise.) A zero value signifies no scaling of the signal, leaving that to some later adjustment (see balance). The default value is 0.

*istor* (optional, default=0) -- initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*kres* - output signal

*ksig* - input signal

*kcf* - the center frequency of the filter, or frequency position of the peak response.

*kbw* - bandwidth of the filter (the Hz difference between the upper and lower half-power points)

*resonxk* is a lot faster than using individual instances in Csound orchestra of the old opcodes, because only one initialization and 'k' cycle are needed at a time, and the audio loop falls entirely inside the cache memory of processor.

## Examples

Here is an example of the resonxk opcode. It uses the file *resonxk.csd* [examples/resonxk.csd].

### Example 824. Example of the resonxk opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o resonxk.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gisin ftgen 0, 0, 2^10, 10, 1

instr 1

ksig randomh 400, 1800, 150
aout poscil .2, 1000+ksig, gisin
outs aout, aout
endin

instr 2

ksig randomh 400, 1800, 150
kcf line 1, p3, 1000 ;vary high-pass
ilay = p4
ksig resonxk ksig, kcf, 100, ilay
aout poscil .2, 1000+ksig, gisin
asig interp ksig ;convert k-rate to a-rate
aout balance asig, aout ;avoid getting asig out of range
outs aout, aout
endin

</CsInstruments>
<CsScore>
i 1 0 5
i 2 6 5 1 ;number of filter stack = 1
i 2 12 5 5 ;number of filter stack = 5
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# resony

`resony` — A bank of second-order bandpass filters, connected in parallel.

## Description

A bank of second-order bandpass filters, connected in parallel.

## Syntax

```
ares resony asig, kbf, kbw, inum, ksep [, isepmode] [, iscl] [, iskip]
```

## Initialization

*inum* -- number of filters

*isepmode* (optional, default=0) -- if *isepmode* = 0, the separation of center frequencies of each filter is generated logarithmically (using octave as unit of measure). If *isepmode* not equal to 0, the separation of center frequencies of each filter is generated linearly (using Hertz). Default value is 0.

*iscl* (optional, default=0) -- coded scaling factor for resonators. A value of 1 signifies a peak response factor of 1, i.e. all frequencies other than *kcf* are attenuated in accordance with the (normalized) response curve. A value of 2 raises the response factor so that its overall RMS value equals 1. (This intended equalization of input and output power assumes all frequencies are physically present; hence it is most applicable to white noise.) A zero value signifies no scaling of the signal, leaving that to some later adjustment (e.g. *balance*). The default value is 0.

*iskip* (optional, default=0) -- initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*asig* -- audio input signal

*kbf* -- base frequency, i.e. center frequency of lowest filter in Hz

*kbw* -- bandwidth in Hz

*ksep* -- separation of the center frequency of filters in octaves

*resony* is a bank of second-order bandpass filters, with k-rate variant frequency separation, base frequency and bandwidth, connected in parallel (i.e. the resulting signal is a mix of the output of each filter). The center frequency of each filter depends of *kbf* and *ksep* variables. The maximum number of filters is set to 100.

## Examples

Here is an example of the *resony* opcode. It uses the file *resony.csd* [examples/resony.csd].

## Example 825. Example of the resonny opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o resonx.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; unfiltered noise

kenv linseg 0, p3*.5, 1, p3*.5, 0 ;envelope
asig rand 0.7 ;white noise
outs asig*kenv, asig*kenv

endin

instr 2 ; filtered noise

ksep = p4 ;vary seperation of center frequency of filters in octaves
kenv linseg 0, p3*.5, 1, p3*.5, 0 ;envelope
asig rand 0.7
kbf line 300, p3, 2000 ;vary base frequency
afilt resonny asig, kbf, 300, 4, ksep
asig balance afilt, asig
outs asig*kenv, asig*kenv

endin
</CsInstruments>
<CsScore>

i 1 0 2
i 2 3 2 1
i 2 6 2 3

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Gabriel Maldonado  
Italy  
1999

New in Csound version 3.56

# resonz

`resonz` — A bandpass filter with variable frequency response.

## Description

Implementations of a second-order, two-pole two-zero bandpass filter with variable frequency response.

## Syntax

```
ares resonz asig, xcf, xbw [, iscl] [, iskip]
```

## Initialization

The optional initialization variables for `resonr` and `resonz` are identical to the i-time variables for `reson`.

`iskip` -- initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

`iscl` -- coded scaling factor for resonators. A value of 1 signifies a peak response factor of 1, i.e. all frequencies other than `kcf` are attenuated in accordance with the (normalized) response curve. A value of 2 raises the response factor so that its overall RMS value equals 1. This intended equalization of input and output power assumes all frequencies are physically present; hence it is most applicable to white noise. A zero value signifies no scaling of the signal, leaving that to some later adjustment (see `balance`). The default value is 0.

## Performance

`resonr` and `resonz` are variations of the classic two-pole bandpass resonator (`reson`). Both filters have two zeroes in their transfer functions, in addition to the two poles. `resonz` has its zeroes located at  $z = 1$  and  $z = -1$ . `resonr` has its zeroes located at  $+\sqrt{R}$  and  $-\sqrt{R}$ , where  $R$  is the radius of the poles in the complex  $z$ -plane. The addition of zeroes to `resonr` and `resonz` results in the improved selectivity of the magnitude response of these filters at cutoff frequencies close to 0, at the expense of less selectivity of frequencies above the cutoff peak.

`resonr` and `resonz` are very close to constant-gain as the center frequency is swept, resulting in a more efficient control of the magnitude response than with traditional two-pole resonators such as `reson`.

`resonr` and `resonz` produce a sound that is considerably different from `reson`, especially for lower center frequencies; trial and error is the best way of determining which resonator is best suited for a particular application.

`asig` -- input signal to be filtered

`xcf` -- cutoff or resonant frequency of the filter, measured in Hz

`xbw` -- bandwidth of the filter (the Hz difference between the upper and lower half-power points)

## Technical History

`resonr` and `resonz` were originally described in an article by Julius O. Smith and James B. Angell.<sup>1</sup> Smith and Angell recommended the `resonz` form (zeros at +1 and -1) when computational efficiency was the

main concern, as it has one less multiply per sample, while *resonr* (zeroes at + and - the square root of the pole radius R) was recommended for situations when a perfectly constant-gain center peak was required.

Ken Steiglitz, in a later article <sup>2</sup>, demonstrated that *resonz* had constant gain at the true peak of the filter, as opposed to *resonr*, which displayed constant gain at the pole angle. Steiglitz also recommended *resonz* for its sharper notches in the gain curve at zero and Nyquist frequency. Steiglitz's recent book <sup>3</sup> features a thorough technical discussion of *reson* and *resonz*, while Dodge and Jerse's textbook <sup>4</sup> illustrates the differences in the response curves of *reson* and *resonz*.

## Examples

Here is an example of the *resonr* and *resonz* opcodes. It uses the file *resonr.csd* [examples/resonr.csd].

### Example 826. Example of the *resonr* and *resonz* opcodes.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc    -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o resonr.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

/* Written by Sean Costello */
; Orchestra file for resonant filter sweep of a sawtooth-like waveform.
; The outputs of reson, resonr, and resonz are scaled by coefficients
; specified in the score, so that each filter can be heard on its own
; from the same instrument.

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 1

idur      =      p3
ibegfreq =      p4          ; beginning of sweep frequency
iendfreq =      p5          ; ending of sweep frequency
ibw       =      p6          ; bandwidth of filters in Hz
ifreq     =      p7          ; frequency of gbuzz that is to be filtered
iamp      =      p8          ; amplitude to scale output by
ires      =      p9          ; coefficient to scale amount of reson in output
iresr     =      p10         ; coefficient to scale amount of resonr in output
iresz     =      p11         ; coefficient to scale amount of resonz in output

; Frequency envelope for reson cutoff
kfreq    linseg ibegfreq, idur * .5, iendfreq, idur * .5, ibegfreq

; Amplitude envelope to prevent clicking
kenv     linseg 0, .1, iamp, idur - .2, iamp, .1, 0

; Number of harmonics for gbuzz scaled to avoid aliasing
iharms   =      (sr*.4)/ifreq

asig     gbuzz 1, ifreq, iharms, 1, .9, 1          ; "Sawtooth" waveform
ain      =      kenv * asig                       ; output scaled by amp envelope
```

```
ares    reson ain, kfreq, ibw, 1
aresr   resonr ain, kfreq, ibw, 1
aresz   resonz ain, kfreq, ibw, 1

        out ares * ires + aresr * iresr + aresz * iresz

endin

</CsInstruments>
<CsScore>

/* Written by Sean Costello */
f1 0 8192 9 1 1 .25           ; cosine table for gbuzz generator

i1 0 10 1 3000 200 100 4000 1 0 0 ; reson output with bw = 200
i1 10 10 1 3000 200 100 4000 0 1 0 ; resonr output with bw = 200
i1 20 10 1 3000 200 100 4000 0 0 1 ; resonz output with bw = 200
i1 30 10 1 3000 50 200 8000 1 0 0 ; reson output with bw = 50
i1 40 10 1 3000 50 200 8000 0 1 0 ; resonr output with bw = 50
i1 50 10 1 3000 50 200 8000 0 0 1 ; resonz output with bw = 50
e

</CsScore>
</CsoundSynthesizer>
```

## References

1. Smith, Julius O. and Angell, James B., "A Constant-Gain Resonator Tuned by a Single Coefficient," *Computer Music Journal*, vol. 6, no. 4, pp. 36-39, Winter 1982.
2. Steiglitz, Ken, "A Note on Constant-Gain Digital Resonators," *Computer Music Journal*, vol. 18, no. 4, pp. 8-10, Winter 1994.
3. Ken Steiglitz, *A Digital Signal Processing Primer, with Applications to Digital Audio and Computer Music*. Addison-Wesley Publishing Company, Menlo Park, CA, 1996.
4. Dodge, Charles and Jerse, Thomas A., *Computer Music: Synthesis, Composition, and Performance*. New York: Schirmer Books, 1997, 2nd edition, pp. 211-214.

## See Also

*resonr*

## Credits

Author: Sean Costello  
Seattle, Washington  
1999

New in Csound version 3.55

Audio rate parameters introduced in version 6.02

November 2013.

# resyn

resyn — Streaming partial track additive synthesis with cubic phase interpolation with pitch control and support for timescale-modified input

## Description

The resyn opcode takes an input containing a TRACKS pv streaming signal (as generated, for instance by partials). It resynthesises the signal using linear amplitude and cubic phase interpolation to drive a bank of interpolating oscillators with amplitude and pitch scaling controls. Resyn is a modified version of sinsyn, allowing for the resynthesis of data with pitch and timescale changes.

## Syntax

```
asig resyn fin, kscal, kpitch, kmaxtracks, ifn
```

## Performance

*asig* -- output audio rate signal

*fin* -- input pv stream in TRACKS format

*kscal* -- amplitude scaling

*kpitch* -- pitch scaling

*kmaxtracks* -- max number of tracks in resynthesis. Limiting this will cause a non-linear filtering effect, by discarding newer and higher-frequency tracks (tracks are ordered by start time and ascending frequency, respectively)

*ifn* -- function table containing one cycle of a sinusoid (sine or cosine)

## Examples

Here is an example of the resyn opcode. It uses the file *resyn.csd* [examples/resyn.csd].

### Example 827. Example of the resyn opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o resyn.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
```

```
nchnls = 2
odbfs = 1

instr 1

ktracks = p4
ain diskin2 "fox.wav", 1, 0, 1
fsl,fsi2 pvsifd ain, 2048, 512, 1 ; pvsifd analysis
fst partials fs1, fsi2, .1, 1,3, 500 ; partial tracking
aout resyn fst, 1, 1.5, ktracks, 1 ; resynthesis (up a 5th)
outs aout, aout

endin
</CsInstruments>
<CsScore>
;sine
f1 0 4096 10 1

i 1 0 2.7 500
i 1 3 2.7 10 ;non-linear filtering effect

e
</CsScore>
</CsoundSynthesizer>
```

The example above shows partial tracking of an ifd-analysis signal and cubic-phase additive resynthesis with pitch shifting.

## Credits

Author: Victor Lazzarini  
June 2005

New plugin in version 5

November 2004.

## return

return — Returns a value from an instrument.

## Description

Return will return a value from an instrument at i-time. The value of a global instrument (instrument 0) can be retrieved after compilation by the evalstr opcode. The retrieval of values returned by other instruments is not yet implemented.

## Syntax

```
return ival
```

## Initialization

*ival* -- a value to be returned by instrument.

## Examples

Here is an example of the return opcode in conjunction with evalcode:

### Example 828.

```
ival evalstr "return 2 + 2"  
print ival
```

## Credits

Author: Victor Lazzarini, 2013

# reverb

reverb — Reverberates an input signal with a “natural room” frequency response.

## Description

Reverberates an input signal with a “natural room” frequency response.

## Syntax

```
ares reverb asig, krvt [, iskip]
```

## Initialization

*iskip* (optional, default=0) -- initial disposition of delay-loop data space (cf. *reson*). The default value is 0.

## Performance

*krvt* -- the reverberation time (defined as the time in seconds for a signal to decay to 1/1000, or 60dB down from its original amplitude).

A standard *reverb* unit is composed of four *comb* filters in parallel followed by two *alpass* units in series. Loop times are set for optimal “natural room response.” Core storage requirements for this unit are proportional only to the sampling rate, each unit requiring approximately 3K words for every 10KC. The *comb*, *alpass*, *delay*, *tone* and other Csound units provide the means for experimenting with alternate reverberator designs.

Since output from the standard *reverb* will begin to appear only after 1/20 second or so of delay, and often with less than three-fourths of the original power, it is normal to output both the source and the reverberated signal. If *krvt* is inadvertently set to a non-positive number, *krvt* will be reset automatically to 0.01. (New in Csound version 4.07.) Also, since the reverberated sound will persist long after the cessation of source events, it is normal to put *reverb* in a separate instrument to which sound is passed via a *global variable*, and to leave that instrument running throughout the performance.

## Examples

Here is an example of the reverb opcode. It uses the file *reverb.csd* [examples/reverb.csd].

### Example 829. Example of the reverb opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;;realtime audio out
;-iadc   ;;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o reverb.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>
```

```
sr = 44100
ksmps = 32
Odbfs = 1
nchnls = 2

gal init 0

instr 1

asig poscil .2, cpspch(p4), 1
outs asig, asig

gal += asig ;add direct signal to global reverb

endin

instr 99 ;(highest instr number executed last)

arev reverb gal, 1.5
outs arev, arev

gal = 0 ;clear
endin

</CsInstruments>
<CsScore>
f 1 0 128 10 1 ;sine

i 1 0 0.1 7.00 ;short sounds
i 1 1 0.1 8.02
i 1 2 0.1 8.04
i 1 3 0.1 8.06

i 99 0 6 ;reverb runs for 6 seconds
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*alpass, comb, valpass, vcomb*

## Credits

Author: William "Pete" Moss  
University of Texas at Austin  
Austin, Texas USA  
January 2002

## reverb2

reverb2 — Same as the nreverb opcode.

### Description

Same as the *nreverb* opcode.

### Syntax

```
ares reverb2 asig, ktime, khdif [, iskip] [,inumCombs] \  
    [, ifnCombs] [, inumAlpas] [, ifnAlpas]
```

# reverbsc

reverbsc — 8 delay line stereo FDN reverb, based on work by Sean Costello

## Description

8 delay line stereo FDN reverb, with feedback matrix based upon physical modeling scattering junction of 8 lossless waveguides of equal characteristic impedance. Based on Csound orchestra version by Sean Costello.

## Syntax

```
aoutL, aoutR reverbsc ainL, ainR, kfblvl, kfco[, israte[, ipitchm[, iskip]]]
```

## Initialization

*israte* (optional, defaults to the orchestra sample rate) -- assume a sample rate of *israte*. This is normally set to *sr*, but a different setting can be useful for special effects.

*ipitchm* (optional, defaults to 1) -- depth of random variation added to delay times, in the range 0 to 10. The default is 1, but this may be too high and may need to be reduced for held pitches such as piano tones.

*iskip* (optional, defaults to zero) -- if non-zero, initialization of the opcode is skipped, whenever possible.

## Performance

*aoutL*, *aoutR* -- output signals for left and right channel

*ainL*, *ainR* -- left and right channel input. Note that having an input signal on either the left or right channel only will still result in having reverb output on both channels, making this unit more suitable for reverberating stereo input than the *freeverb* opcode.

*kfblvl* -- feedback level, in the range 0 to 1. 0.6 gives a good small "live" room sound, 0.8 a small hall, and 0.9 a large hall. A setting of exactly 1 means infinite length, while higher values will make the opcode unstable.

*kfco* -- cutoff frequency of simple first order lowpass filters in the feedback loop of delay lines, in Hz. Should be in the range 0 to  $israte/2$  (not  $sr/2$ ). A lower value means faster decay in the high frequency range.

## Examples

Here is an example of the *reverbsc* opcode. It uses the file *reverbsc.csd* [examples/reverbsc.csd].

### Example 830. An example of the reverbsc opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform
```

```
; Audio out  Audio in  No messages
-odac        -iadc     -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o reverb.sc.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
sr         = 48000
ksmps     = 32
nchnls    = 2
0dbfs     = 1

          instr 1
a1        vco2 0.85, 440, 10
kfrq      port 100, 0.004, 20000
a1        butterlp a1, kfrq
a2        linseg 0, 0.003, 1, 0.01, 0.7, 0.005, 0, 1, 0
a1        = a1 * a2
a2        = a1 * p5
a1        = a1 * p4
          denorm a1, a2
aL, aR    reverb.sc a1, a2, 0.85, 12000, sr, 0.5, 1
          outs a1 + aL, a2 + aR
          endin

</CsInstruments>
<CsScore>
i 1 0 1 0.71 0.71
i 1 1 1 0 1
i 1 2 1 -0.71 0.71
i 1 3 1 1 0
i 1 4 4 0.71 0.71
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Istvan Varga  
2005

# rewindscore

rewindscore — Rewinds the playback position of the current score performance.

## Description

Rewinds the playback position of the current score performance..

## Syntax

```
rewindscore
```

## Examples

Here is an example of the *rewindscore* opcode. It uses the file *rewindscore.csd* [examples/rewindscore.csd].

### Example 831. An example of the rewindscore opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o rewindscore.wav -W   ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kenv expon 1, p3, 0.0001
aout poscil .5*kenv, cpspch(p4), 1
outs aout, aout
endin

instr 100

rewindscore

endin
</CsInstruments>
<CsScore>
f1 0 4096 10 1 ;sine wave

i1 0 1 8.00
i1 + 1 8.03
i1 + 1 8.04
i1 + 1 8.07
```

```
i1 + 1 8.09
i1 + 1 8.10
i1 + 1 8.09
i1 + 1 8.07
i1 + 1 8.03

i100 9 1 ;rewind from 9th second
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*setscorepos*

## Credits

Author: Victor Lazzarini  
2008

New in Csound version 5.09

# rezzy

rezzy — A resonant low-pass filter.

## Description

A resonant low-pass filter.

## Syntax

```
ares rezzy asig, xfco, xres [, imode, iskip]
```

## Initialization

*imode* (optional, default=0) -- high-pass or low-pass mode. If zero, *rezzy* is low-pass. If not zero, *rezzy* is high-pass. Default value is 0. (New in Csound version 3.50)

*iskip* (optional, default=0) -- if non zero skip the initialisation of the filter. (New in Csound version 4.23f13 and 5.0)

## Performance

*asig* -- input signal

*xfco* -- filter cut-off frequency in Hz. As of version 3.50, may i-,k-, or a-rate.

*xres* -- amount of resonance. Values of 1 to 100 are typical. Resonance should be one or greater. As of version 3.50, may a-rate, i-rate, or k-rate.

*rezzy* is a resonant low-pass filter created empirically by Hans Mikelson.

## Examples

Here is an example of the *rezzy* opcode. It uses the file *rezzy.csd* [examples/rezzy.csd].

### Example 832. Example of the *rezzy* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o rezzy.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
0dbfs = 1
```

```
nchnls = 2

instr 1

asaw vco2 .3, 110 ;sawtooth
kcf line 1760, p3, 220 ;vary cut-off frequency from 220 to 1280 Hz
kres = p4 ;vary resonance too
ares rezzy asaw, kcf, kres
asig balance ares, asaw
outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 4 10
i 1 + 4 30
i 1 + 4 120 ;lots of resonance
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*biquad, moogvcf*

## Credits

Author: Hans Mikelson  
October 1998

New in Csound version 3.49

# rfft

rfft — Fast Fourier Transform of a real-value array.

## Description

Applies a forward Fast Fourier Transform to a real-value input 1-dimensional array. The output is another array containing the transform, non-redundant, non-negative spectrum only. If the input array is power-of-two, the output array size will match the input size, with the first two points containing 0Hz and Nyquist frequency coefficients. Otherwise, the output will have two extra values (input size + 2), and the the Nyquist coefficient will be placed at `kin[input_size]` position (`kin[1]` and `kin[input_size+1]` will be 0).

## Syntax

```
kout[] rfft kin[]
```

## Performance

`kout[]` -- output array containing the transform. It will be created if it does not exist.

`kin[]` -- input array containing the real-valued input.

## Examples

Here is an example of the rfft opcode. It uses the file `rfft.csd` [examples/rfft.csd].

### Example 833. Example of the rfft opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>

<CsOptions>
-d -o dac
</CsOptions>

<CsInstruments>
;ksmps needs to be an integer div of hopsize
ksmps = 64

instr 1

  ihopsize = 256 ; hopsize
  ifftsize = 1024 ; FFT size
  iolaps = ifftsize/ihopsize ; overlaps
  ibw = sr/ifftsize ; bin bandwidth
  kcnt init 0 ; counting vars
  krow init 0

  kOla[] init ifftsize ; overlap-add buffer
  kIn[] init ifftsize ; input buffer
  kSw[] init ifftsize
  kOut[][] init iolaps, ifftsize ; output buffers
```

```

a1 diskin2 "fox.wav",1,0,1 ; audio input
ks expon 100, p3, 1000
asw vco2 k(a1), ks

/* every hopsize samples */
if kcnt == ihopsize then
  /* window and take FFT */
  kWin[] window kIn,krow*ihopsize
  kSpec[] rfft kWin
  kWin window kSw,krow*ihopsize
  kSpec2[] rfft kWin
  kProd[] cmplxprod kSpec, kSpec2

  /* IFFT + window */
  kRow[] rifft kProd
  kWin window kRow, krow*ihopsize
  /* place it on out buffer */
  kOut setrow kWin, krow

  /* zero the ola buffer */
  kOla = 0
  /* overlap-add */
  ki = 0
  until ki == iolaps do
    kRow getrow kOut, ki
    kOla = kOla + kRow
    ki += 1
  od

  /* update counters */
  krow = (krow+1)%iolaps
  kcnt = 0
endif

/* shift audio in/out of buffers */
kIn shiftin a1
kSw shiftin asw
a2 shiftout kOla
  out a2/iolaps

/* increment counter */
kcnt += ksmps

endin

</CsInstruments>

<CsScore>
i1 0 10
</CsScore>

</CsoundSynthesizer>

```

## See Also

Vectorial opcodes, array opcodes

## Credits

Author: Victor Lazzarini  
 NUI Maynooth  
 2014

New in version 6.04

# rifft

rifft — Complex-to-real Inverse Fast Fourier Transform.

## Description

Applies an Inverse Fast Fourier Transform to a complex-value input 1-dimensional array producing a real-valued output. The output is another array containing the real-valued signal. If the input array is power-of-two, the output array size will match the input size. Otherwise, the output will have two fewer values (input size - 2).

## Syntax

```
kout[] rifft kin[]
```

## Performance

*kout[]* -- output array containing the real-valued output. It will be created if it does not exist.

*kin[]* -- input array containing the complex input.

## Examples

Here is an example of the rifft opcode. It uses the file *irfft.csd* [examples/irfft.csd].

### Example 834. Example of the rifft opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>

<CsOptions>
-d -o dac
</CsOptions>

<CsInstruments>
;ksmps needs to be an integer div of hopsize
ksmps = 64

instr 1

  ihopsize = 256 ; hopsize
  ifftsize = 1024 ; FFT size
  iolaps = ifftsize/ihopsize ; overlaps
  ibw = sr/ifftsize ; bin bandwidth
  kcmt init 0 ; counting vars
  krow init 0

  kOla[] init ifftsize ; overlap-add buffer
  kIn[] init ifftsize ; input buffer
  kOut[][] init iolaps, ifftsize ; output buffers

  a1 diskin2 "fox.wav",1,0,1 ; audio input

  /* every hopsize samples */
```

```

if kcnt == ihopsize then
  /* window and take FFT */
  kWin[] window kIn,krow*ihopsize
  kSpec[] rfft kWin

  /* filter between high and low freqs */
  ilow = 0
  ihigh = 1000
  ki = int(ilow/ibw)
  until ki == int(ihigh/ibw) do
    kSpec[ki] = 0
    ki += 1
  od

  /* IFFT + window */
  kRow[] rfft kSpec
  kWin window kRow, krow*ihopsize
  /* place it on out buffer */
  kOut setrow kWin, krow

  /* zero the ola buffer */
  kOla = 0
  /* overlap-add */
  ki = 0
  until ki == iolaps do
    kRow getrow kOut, ki
    kOla = kOla + kRow
    ki += 1
  od

  /* update counters */
  krow = (krow+1)%iolaps
  kcnt = 0
endif

/* shift audio in/out of buffers */
kIn shiftin a1
a2 shiftout kOla
  out a2/iolaps

/* increment counter */
kcnt += ksmpls

endin

</CsInstruments>

<CsScore>
i1 0 10
</CsScore>

</CsoundSynthesizer>

```

## See Also

Vectorial opcodes, array opcodes

## Credits

Author: Victor Lazzarini  
 NUI Maynooth  
 2014

New in version 6.04

# rigoto

rigoto — Transfers control during a reinit pass.

## Description

Similar to *igoto*, but effective only during a *reinit* pass (i.e., no-op at standard i-time). This statement is useful for bypassing units that are not to be reinitialized.

## Syntax

```
rigoto label
```

## See Also

*cigoto, igoto, reinit, rireturn*

# rireturn

rireturn — Terminates a reinit pass.

## Description

Terminates a *reinit* pass (i.e., no-op at standard i-time). This statement, or an *endin*, will cause normal performance to be resumed.

## Syntax

```
rireturn
```

## Examples

The following statements will generate an exponential control signal whose value moves from 440 to 880 exactly ten times over the duration *p3*. They use the file *reinit.csd* [examples/reinit.csd].

### Example 835. Example of the rireturn opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o reinit.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

instr 1

reset:
  timeout 0, p3/10, contin
  reinit reset

contin:
  kLine expon 440, p3/10, 880
  aSig oscil 10000, kLine, 1
  out aSig
  rireturn

endin

</CsInstruments>
<CsScore>

f1 0 4096 10 1
```

```
i1 0 10  
e
```

```
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*reinit, rigoto*

## rms

rms — Determines the root-mean-square amplitude of an audio signal.

### Description

Determines the root-mean-square amplitude of an audio signal. It low-pass filters the actual value, to average in the manner of a VU meter.

### Syntax

```
kres rms asig [, ihp] [, iskip]
```

### Initialization

*ihp* (optional, default=10) -- half-power point (in Hz) of a special internal low-pass filter. The default value is 10.

*iskip* (optional, default=0) -- initial disposition of internal data space (see *reson*). The default value is 0.

### Performance

*asig* -- input audio signal

*kres* -- low-pass filtered rms value of the input signal

*rms* output values *kres* will trace the root-mean-square value of the audio input *asig*. This unit is not a signal modifier, but functions rather as a signal power-gauge. It uses an internal low-pass filter to make the response smoother. *ihp* can be used to control this smoothing. The higher the value, the "snappier" the measurement.

This opcode can also be used as an envelope follower.

The *kres* output from this opcode is given in amplitude and depends on *Odbfs*. If you want the output in decibels, you can use *dbamp*

### Examples

```
arms rms    asig ; get rms value of signal asig
```

Here is an example of the rms opcode. It uses the file *rms.csd* [examples/rms.csd].

#### Example 836. Example of the rms opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform
```

```

; Audio out   Audio in   No messages
-odac        -iadc      -d -m0   ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o rms.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
ksmps = 128
nchnls = 1

;Example by Andres Cabrera 2007

Odbfs = 1

FLpanel "rms", 400, 100, 50, 50
  gkrmstext, gihrmstext FLtext "Rms", -100, 0, 0.1, 3, 110, 30, 60, 50
  gkihp, gihandle FLtext "ihp", 0, 10, 0.05, 1, 100, 30, 220, 50
  gkrmsslider, gihrmsslider FLslider "", -60, -0.5, -1, 5, -1, 380, 20, 10, 10

FLpanelEnd
FLrun

FLsetVal_i 5, gihandle
; Instrument #1.
instr 1
  a1 inch 1

label:
  kval rms a1, i(gkihp) ;measures rms of input channel 1
  rireturn

  kval = dbamp(kval) ; convert to db full scale
  printk 0.5, kval
  FLsetVal 1, kval, gihrmsslider ;update the slider and text values
  FLsetVal 1, kval, gihrmstext
  knewihp changed gkihp ; reinit when ihp text has changed
  if (knewihp == 1) then
    reinit label ;needed because ihp is an i-rate parameter
  endif
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one minute
i 1 0 60
e

</CsScore>
</CsoundSynthesizer>

```

## See Also

*balance, gain*

# rnd

rnd — Returns a random number in a unipolar range at the rate given by the input argument.

## Description

Returns a random number in a unipolar range at the rate given by the input argument.

## Syntax

```
rnd(x) (init- or control-rate only)
```

Where the argument within the parentheses may be an expression. These value converters sample a global random sequence, but do not reference *seed*. The result can be a term in a further expression.

## Performance

Returns a random number in the unipolar range 0 to *x*.

## Examples

Here is an example of the rnd opcode. It uses the file *rnd.csd* [examples/rnd.csd].

### Example 837. Example of the rnd opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o rnd.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Andres Cabrera 2010

sr = 44100
ksmps = 4410
nchnls = 1
0dbfs = 1

instr 1
; Generate a random number from 0 to 10.
irand = rnd(10)
print irand
endin

instr 2
klimit init 10
krand = rnd(klimit)
printk 0, krand
endin
```

```
</CsInstruments>
<CsScore>

i 1 0 1 ; Generate 1 number
i 1 0 1 ; Generate another number
i 1 0 1 ; yet another number

i 2 2 1 ; 1 second prints 9 values (kr = 10)
e

</CsScore>
</CsoundSynthesizer>
```

Its output should be:

```
SECTION 1:
new alloc for instr 1:
instr 1: irand = 9.735
new alloc for instr 1:
instr 1: irand = 1.394
new alloc for instr 1:
instr 1: irand = 7.695
midi channel 1 now using instr 1
B 0.000 .. 2.000 T 2.000 TT 2.000 M: 0.00000
new alloc for instr 2:
i 2 time 2.10000: 5.25005
i 2 time 2.20000: 6.22665
i 2 time 2.30000: 9.69511
i 2 time 2.40000: 7.16822
i 2 time 2.50000: 9.45134
i 2 time 2.60000: 1.34123
i 2 time 2.70000: 2.09879
i 2 time 2.80000: 2.36001
i 2 time 2.90000: 0.03553
```

## See Also

*birnd*

## Credits

Author: Barry L. Vercoe  
MIT  
Cambridge, Massachusetts  
1997

# rnd31

rnd31 — 31-bit bipolar random opcodes with controllable distribution.

## Description

31-bit bipolar random opcodes with controllable distribution. These units are portable, i.e. using the same seed value will generate the same random sequence on all systems. The distribution of generated random numbers can be varied at k-rate.

## Syntax

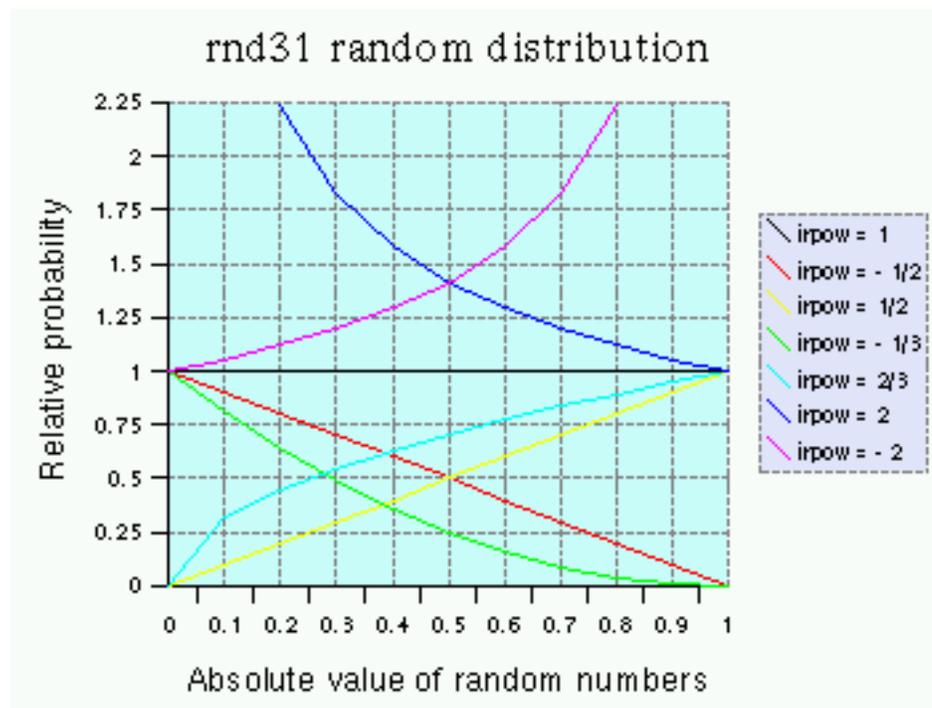
```
ax rnd31 kscl, krpow [, iseed]
ix rnd31 iscl, irpow [, iseed]
kx rnd31 kscl, krpow [, iseed]
```

## Initialization

*ix* -- i-rate output value.

*iscl* -- output scale. The generated random numbers are in the range -iscl to iscl.

*irpow* -- controls the distribution of random numbers. If irpow is positive, the random distribution (x is in the range -1 to 1) is  $abs(x)^{(1/irpow) - 1}$ ; for negative irpow values, it is  $(1 - abs(x))^{(-1/irpow) - 1}$ . Setting *irpow* to -1, 0, or 1 will result in uniform distribution (this is also faster to calculate).



A graph of distributions for different values of irpow.

*iseed* (optional, default=0) -- seed value for random number generator (positive integer in the range 1 to 2147483646 ( $2^{31} - 2$ )). Zero or negative value seeds from current time (this is also the default). Seeding from current time is guaranteed to generate different random sequences, even if multiple random opcodes are called in a very short time.

In the a- and k-rate version the seed is set at opcode initialization. With i-rate output, if *iseed* is zero or negative, it will seed from current time in the first call, and return the next value from the random sequence in successive calls; positive seed values are set at all i-rate calls. The seed is local for a- and k-rate, and global for i-rate units.



## Notes

- although seed values up to 2147483646 are allowed, it is recommended to use smaller numbers (< 1000000) for portability, as large integers may be rounded to a different value if 32-bit floats are used.
- i-rate *rnd31* with a positive seed will always produce the same output value (this is not a bug). To get different values, set seed to 0 in successive calls, which will return the next value from the random sequence.

## Performance

*ax* -- a-rate output value.

*kx* -- k-rate output value.

*kscl* -- output scale. The generated random numbers are in the range -kscl to kscl. It is the same as *iscl*, but can be varied at k-rate.

*krpow* -- controls the distribution of random numbers. It is the same as *irpow*, but can be varied at k-rate.

## Examples

Here is an example of the *rnd31* opcode at a-rate. It uses the file *rnd31.csd* [examples/rnd31.csd].

### Example 838. An example of the *rnd31* opcode at a-rate.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      -iadc      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o rnd31.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
```

```

; Create random numbers at a-rate in the range -2 to 2 with
; a triangular distribution, seed from the current time.
a31 rnd31 2, -0.5

; Use the random numbers to choose a frequency.
afreq = a31 * 500 + 100

a1 oscil 30000, afreq, 1
out a1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
e

</CsScore>
</CsoundSynthesizer>

```

Here is an example of the rnd31 opcode at k-rate. It uses the file *rnd31\_krate.csd* [examples/rnd31\_krate.csd].

### Example 839. An example of the rnd31 opcode at k-rate.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o rnd31_krate.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Create random numbers at k-rate in the range -1 to 1
; with a uniform distribution, seed=10.
k1 rnd31 1, 0, 10

printks "k1=%f\\n", 0.1, k1
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
e

```

```
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
k1=0.112106
k1=-0.274665
k1=0.403933
```

Here is an example of the `rnd31` opcode that uses the number 7 as a seed value. It uses the file `rnd31_seed7.csd` [examples/rnd31\_seed7.csd].

### Example 840. An example of the `rnd31` opcode that uses the number 7 as a seed value.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o rnd31_seed7.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; i-rate random numbers with linear distribution, seed=7.
; (Note that the seed was used only in the first call.)
i1 rnd31 1, 0.5, 7
i2 rnd31 1, 0.5
i3 rnd31 1, 0.5

print i1
print i2
print i3
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
instr 1: i1 = -0.649
instr 1: i2 = -0.761
instr 1: i3 = 0.677
```

Here is an example of the `rnd31` opcode that uses the current time as a seed value. It uses the file `rnd31_time.csd` [examples/rnd31\_time.csd].

**Example 841. An example of the `rnd31` opcode that uses the current time as a seed value.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o rnd31_time.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; i-rate random numbers with linear distribution,
; seeding from the current time. (Note that the seed
; was used only in the first call.)
i1 rnd31 1, 0.5, 0
i2 rnd31 1, 0.5
i3 rnd31 1, 0.5

print i1
print i2
print i3
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
instr 1:  i1 = -0.691
instr 1:  i2 = -0.686
instr 1:  i3 = -0.358
```

## Credits

Author: Istvan Varga

New in version 4.16

# round

`round` — Returns the integer value nearest to  $x$  ; if the fractional part of  $x$  is exactly 0.5, the direction of rounding is undefined.

## Description

The integer value nearest to  $x$  ; if the fractional part of  $x$  is exactly 0.5, the direction of rounding is undefined.

## Syntax

`round(x)` (init-, control-, or audio-rate arg allowed)

`round(k/i[])` (k- or i-arrays )

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

## Examples

Here is an example of the `round` opcode. It uses the file `round.csd` [examples/round.csd].

### Example 842. Example of the round opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
;-o round.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

idiv init 1

loop:
inumber = 9
i1 = inumber / idiv
iro = round(i1)
print inumber, idiv, iro ;print number / idiv = result using round
idiv = idiv + 1
if (idiv <= 10) igoto loop

endin
</CsInstruments>
```

```
<CsScore>
i 1 0 0
e
</CsScore>
</CsoundSynthesizer>
```

Its output should include a line like these:

```
instr 1: inumber = 9.000 idiv = 1.000 ifl = 9.000
instr 1: inumber = 9.000 idiv = 2.000 ifl = 5.000
instr 1: inumber = 9.000 idiv = 3.000 ifl = 3.000
instr 1: inumber = 9.000 idiv = 4.000 ifl = 2.000
instr 1: inumber = 9.000 idiv = 5.000 ifl = 2.000
instr 1: inumber = 9.000 idiv = 6.000 ifl = 2.000
instr 1: inumber = 9.000 idiv = 7.000 ifl = 1.000
instr 1: inumber = 9.000 idiv = 8.000 ifl = 1.000
instr 1: inumber = 9.000 idiv = 9.000 ifl = 1.000
instr 1: inumber = 9.000 idiv = 10.000 ifl = 1.000
```

## See Also

*abs, exp, int, log, log10, i, sqrt*

## Credits

Author: Istvan Varga  
New in Csound 5  
2005

# rspline

rspline — Generate random spline curves.

## Description

Generate random spline curves.

## Syntax

```
ares rspline xrangeMin, xrangeMax, kcpsMin, kcpsMax
kres rspline krangeMin, krangeMax, kcpsMin, kcpsMax
```

## Performance

*kres, ares* -- Output signal

*xrangeMin, xrangeMax* -- Range of values of random-generated points

*kcpsMin, kcpsMax* -- Range of point-generation rate. Min and max limits are expressed in cps.

*rspline* (random-spline-curve generator) is similar to *jspline* but output range is defined by means of two limit values. Also in this case, real output range could be a bit greater of range values, because of interpolating curves between each pair of random-points.

At present time generated curves are quite smooth when *cpsMin* is not too different from *cpsMax*. When *cpsMin-cpsMax* interval is big, some little discontinuity could occur, but it should not be a problem, in most cases. Maybe the algorithm will be improved in next versions.

These opcodes are often better than *jitter* when user wants to “naturalize” or “analogize” digital sounds. They could be used also in algorithmic composition, to generate smooth random melodic lines when used together with *samphold* opcode.

Note that the result is quite different from the one obtained by filtering white noise, and they allow the user to obtain a much more precise control.

## Examples

Here is an example of the *rspline* opcode. It uses the file *rspline.csd* [examples/rspline.csd].

### Example 843. Example of the *rspline* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o jspline.wav -W ;; for file output any platform
```

```
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

krangeMin init p4
krangeMax init p5
kcpsmin init 2
kcpsmax init 3

ksp rspline krangeMin, krangeMax, kcpsmin, kcpsmax
aout pluck 1, 200+ksp, 1000, 0, 1
aout dcblock aout ;remove DC
outs aout, aout

endin
</CsInstruments>
<CsScore>

i 1 0 10 2 5 ;a bit jitter
i 1 8 10 10 20 ;some more
i 1 16 10 20 50 ;lots more
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Gabriel Maldonado

New in version 4.15

# rtclock

rtclock — Read the real time clock from the operating system.

## Description

Read the real-time clock from the operating system.

## Syntax

```
ires rtclock
```

```
kres rtclock
```

## Performance

Read the real-time clock from operating system. Under Windows, this changes only once per second. Under GNU/Linux, it ticks every microsecond. Performance under other systems varies.

## Examples

Here is an example of the rtclock opcode. It uses the file *rtclock.csd* [examples/rtclock.csd].

### Example 844. Example of the rtclock opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n      ;;no sound
;-iadc  ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o rtclock.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

;after an example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

FLcolor 200, 200, 200, 0, 0, 0
; LABEL      | WIDTH | HEIGHT | X | Y
FLpanel "rtclock", 500, 130, 0, 0
;
; ON,OFF,TYPE,WIDTH, HEIGHT, X, Y, OPCODE, INS,START,IDUR
gkOnOff,ihOnOff FLbutton "On/Off", 1, 0, 22, 150, 25, 5, 5, 0, 1, 0, 3600
gkExit,ihExit FLbutton "exitnow",1, 0, 21, 150, 25, 345, 5, 0, 999, 0, 0.001
FLsetColor2 255, 0, 50, ihOnOff ;reddish color

;VALUE DISPLAY BOXES WIDTH,HEIGHT,X, Y
gidclock FLvalue "clock", 100, 25, 200, 60
FLsetVal_i 1, ihOnOff
```

```
FLpanel_end
FLrun

instr 1

if gkOnOff !=0 kgoto CONTINUE ;sense if FLTK on/off switch is not off (in which case skip the next line)
turnoff          ;turn this instr. off now
CONTINUE:
ktime rtclock          ;clock continues to run even
FLprintk2 ktime, gidclock ;after the on/off button was used to stop

endin

instr 999

exitnow          ;exit Csound as fast as possible

endin
</CsInstruments>
<CsScore>

f 0 60 ;runs 60 seconds
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John fitch

New in version 4.10

## S

S — Returns an S-type equivalent of an init-time or k-rate argument.

### Description

Returns a string containing the numeric value of its argument.

### Syntax

```
s(x) (control-rate or init-rate arg)
```

### See Also

*a, k, abs, exp, frac, int, log, log10, sqrt*

### Credits

V Lazzarini, 2016. New in version 6.07

# s16b14

s16b14 — Creates a bank of 16 different 14-bit MIDI control message numbers.

## Description

Creates a bank of 16 different 14-bit MIDI control message numbers.

## Syntax

```
i1,...,i16 s16b14 ichan, ictlno_msb1, ictlno_lsb1, imin1, imax1, \  
    initvalue1, ifn1,..., ictlno_msb16, ictlno_lsb16, imin16, imax16, initvalue16, ifn16  
k1,...,k16 s16b14 ichan, ictlno_msb1, ictlno_lsb1, imin1, imax1, \  
    initvalue1, ifn1,..., ictlno_msb16, ictlno_lsb16, imin16, imax16, initvalue16, ifn16
```

## Initialization

*i1 ... i64* -- output values

*ichan* -- MIDI channel (1-16)

*ictlno\_msb1 ... ictlno\_msb32* -- MIDI control number, most significant byte (0-127)

*ictlno\_lsb1 ... ictlno\_lsb32* -- MIDI control number, least significant byte (0-127)

*imin1 ... imin64* -- minimum values for each controller

*imax1 ... imax64* -- maximum values for each controller

*init1 ... init64* -- initial value for each controller

*ifn1 ... ifn64* -- function table for conversion for each controller

## Performance

*k1 ... k64* -- output values

*s16b14* is a bank of MIDI controllers, useful when using MIDI mixer such as Kawai MM-16 or others for changing whatever sound parameter in real-time. The raw MIDI control messages at the input port are converted to agree with *iminN* and *imaxN*, and an initial value can be set. Also, an optional non-interpolated function table with a custom translation curve is allowed, useful for enabling exponential response curves.

When no function table translation is required, set the *ifnN* value to 0, else set *ifnN* to a valid function table number. When table translation is enabled (i.e. setting *ifnN* value to a non-zero number referring to an already allocated function table), *initN* value should be set equal to *iminN* or *imaxN* value, else the initial output value will not be the same as specified in *initN* argument.

*s16b14* allows a bank of 16 different MIDI control message numbers. It uses 14-bit values instead of MIDI's normal 7-bit values.

As the input and output arguments are many, you can split the line using `\` (backslash) character (new in 3.47 version) to improve the readability. Using these opcodes is considerably more efficient than using the separate ones (*ctrl7* and *tonek*) when more controllers are required.

In the i-rate version of *s16b14*, there is not an initial value input argument. The output is taken directly from the current status of internal controller array of Csound.

## Credits

Author: Gabriel Maldonado  
Italy  
December 1998

New in Csound version 3.50

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# s32b14

s32b14 — Creates a bank of 32 different 14-bit MIDI control message numbers.

## Description

Creates a bank of 32 different 14-bit MIDI control message numbers.

## Syntax

```
i1,...,i32 s32b14 ichan, ictrlno_msb1, ictrlno_lsb1, imin1, imax1, \  
    initvalue1, ifn1,..., ictrlno_msb32, ictrlno_lsb32, imin32, imax32, initvalue32, ifn32  
  
k1,...,k32 s32b14 ichan, ictrlno_msb1, ictrlno_lsb1, imin1, imax1, \  
    initvalue1, ifn1,..., ictrlno_msb32, ictrlno_lsb32, imin32, imax32, initvalue32, ifn32
```

## Initialization

*i1 ... i64* -- output values

*ichan* -- MIDI channel (1-16)

*ictrlno\_msb1 ... ictrlno\_msb32* -- MIDI control number, most significant byte (0-127)

*ictrlno\_lsb1 ... ictrlno\_lsb32* -- MIDI control number, least significant byte (0-127)

*imin1 ... imin64* -- minimum values for each controller

*imax1 ... imax64* -- maximum values for each controller

*init1 ... init64* -- initial value for each controller

*ifn1 ... ifn64* -- function table for conversion for each controller

## Performance

*k1 ... k64* -- output values

*s32b14* is a bank of MIDI controllers, useful when using MIDI mixer such as Kawai MM-16 or others for changing whatever sound parameter in real-time. The raw MIDI control messages at the input port are converted to agree with *iminN* and *imaxN*, and an initial value can be set. Also, an optional non-interpolated function table with a custom translation curve is allowed, useful for enabling exponential response curves.

When no function table translation is required, set the *ifnN* value to 0, else set *ifnN* to a valid function table number. When table translation is enabled (i.e. setting *ifnN* value to a non-zero number referring to an already allocated function table), *initN* value should be set equal to *iminN* or *imaxN* value, else the initial output value will not be the same as specified in *initN* argument.

*s32b14* allows a bank of 32 different MIDI control message numbers. It uses 14-bit values instead of MIDI's normal 7-bit values.

As the input and output arguments are many, you can split the line using `\` (backslash) character (new in 3.47 version) to improve the readability. Using these opcodes is considerably more efficient than using the separate ones (*ctrl7* and *tonek*) when more controllers are required.

In the i-rate version of *s32b14*, there is not an initial value input argument. The output is taken directly from the current status of internal controller array of Csound.

## Credits

Author: Gabriel Maldonado  
Italy  
December 1998

New in Csound version 3.50

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# samphold

samphold — Performs a sample-and-hold operation on its input.

## Description

Performs a sample-and-hold operation on its input.

## Syntax

```
ares samphold asig, agate [, ival] [, ivstor]
kres samphold ksig, kgate [, ival] [, ivstor]
```

## Initialization

*ival*, *ivstor* (optional) -- controls initial disposition of internal save space. If *ivstor* is zero the internal “hold” value is set to *ival* ; else it retains its previous value. Defaults are 0,0 (i.e. init to zero)

## Performance

*kgate*, *xgate* -- controls whether to hold the signal.

*samphold* performs a sample-and-hold operation on its input according to the value of *gate*. If *gate*  $\neq 0$ , the input samples are passed to the output; if *gate* = 0, the last output value is repeated. The controlling *gate* can be a constant, a control signal, or an audio signal.

## Examples

```
asrc buzz      10000, 440, 20, 1      ; band-limited pulse train
adif diff      asrc                  ; emphasize the highs
anew balance   adif, asrc             ; but retain the power
agate reson    asrc, 0, 440           ; use a lowpass of the original
asamp samphold anew, agate           ; to gate the new audiosig
aout tone      asamp, 100             ; smooth out the rough edges
```

Here is another example of the samphold opcode. It uses the file *samphold.csd* [examples/samphold.csd].

### Example 845. Example of the samphold opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o samphold.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kx line -1, p3, 1 ; between -1 and +1
ktrig metro 1 ; triggers 1 time per second
kval samphold kx, ktrig ; change value whenever ktrig = 1
    printk2 kval ; will print every time kval changes
asig disk2 "flute.aiff", 1+kval, 0, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 11
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*diff, downsamp, integ, interp, upsamp*

# sandpaper

sandpaper — Semi-physical model of a sandpaper sound.

## Description

*sandpaper* is a semi-physical model of a sandpaper sound. It is one of the PhISEM percussion opcodes. PhISEM (Physically Informed Stochastic Event Modeling) is an algorithmic approach for simulating collisions of multiple independent sound producing objects.

## Syntax

```
ares sandpaper iamp, idettack [, inum] [, idamp] [, imaxshake]
```

## Initialization

*iamp* -- Amplitude of output. Note: As these instruments are stochastic, this is only a approximation.

*idettack* -- period of time over which all sound is stopped

*inum* (optional) -- The number of beads, teeth, bells, timbrels, etc. If zero, the default value is 128.

*idamp* (optional) -- the damping factor, as part of this equation:

$$\text{damping\_amount} = 0.998 + (\text{idamp} * 0.002)$$

The default *damping\_amount* is 0.999 which means that the default value of *idamp* is 0.5. The maximum *damping\_amount* is 1.0 (no damping). This means the maximum value for *idamp* is 1.0.

The recommended range for *idamp* is usually below 75% of the maximum value.

*imaxshake* (optional) -- amount of energy to add back into the system. The value should be in range 0 to 1.

## Examples

Here is an example of the sandpaper opcode. It uses the file *sandpaper.csd* [examples/sandpaper.csd].

### Example 846. Example of the sandpaper opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o sandpaper.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
```

```
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

idmp = p4
a1 line 2, p3, 2 ;preset amplitude increase
a2 sandpaper 1, 0.01, 128, idmp ;sandpaper needs a little amp help at these settings
asig product a1, a2 ;increase amplitude
outs asig, asig

endin
</CsInstruments>
<CsScore>
i1 0 1 0.5
i1 + 1 0.95

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*cabasa, crunch, sekere, stix*

## Credits

Author: Perry Cook, part of the PhOLIES (Physically-Oriented Library of Imitated Environmental Sounds)  
Adapted by John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 4.07

Added notes by Rasmus Ekman on May 2002.

# scale

scale — Arbitrary signal scaling.

## Description

Scales incoming value to user-definable range. Similar to scale object found in popular dataflow languages.

## Syntax

```
ksc1 scale kinput, kmax, kmin
```

## Performance

*kinput* -- Input value. Can originate from any k-rate source as long as that source's output is in range 0-1.

*kmin* -- Minimum value of the resultant scale operation.

*kmax* -- Maximum value of the resultant scale operation.

## Examples

Here is an example of the scale opcode. It uses the file *scale.csd* [examples/scale.csd].

### Example 847. Example of the scale opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  Silent
-odac      -iadc      -d      ;;realtime output
</CsOptions>
<CsInstruments>

sr = 22050
ksmps = 10
nchnls = 2

/*--- */

instr 1 ; scale test

kmod ctrl7 1, 1, 0, 1
printk2 kmod

kout scale kmod, 0, -127
printk2 kout

endin

/*--- */
</CsInstruments>
<CsScore>
```

```
i1 0 8888
```

```
e  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*gainslider, logcurve, expcurve*

## Credits

Author: David Akbari  
October  
2006

# scalearray

scalearray — Scales the values in a range of a vector (one dimensional array).

## Description

The *scalearray* opcode scales a subregion of a vector to a given minimum/maximum.

## Syntax

```
scalearray tarray, kmin, kmax[, kleft, kright]
```

## Performance

*karray* -- array for operation.

*kmin*, *kmax* -- target minimum and maximum values.

*kleft*, *kright* -- range of table to use, defaulting to 0 and size of the vector.

## Examples

Here is an example of the *scalearray* opcode. It uses the file *scalearray.csd* [examples/scalearray.csd].

### Example 848. Example of the *scalearray* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n
</CsOptions>
<CsInstruments>
;example by joachim heintz

          seed      0

instr 1
;create an array with 10 elements
kArr[]    init      10
;fill in random numbers and print them out
          printks   "kArr in maximum range 0..100:\n", 0
kIndx    =          0
          until kIndx == 10 do
kNum     random    0, 100
kArr[kIndx] =      kNum
          printf    "kArr[%d] = %10f\n", kIndx+1, kIndx, kNum
kIndx    +=        1
          od
;scale numbers 0...1 and print them out again
          scalearray kArr, 0, 1
kIndx    =          0
          printks   "kArr in range 0..1\n", 0
          until kIndx == 10 do
```

```

kIndx      printf      "kArr[%d] = %10f\n", kIndx+1, kIndx, kArr[kIndx]
           +=          1
           od
           turnoff
endin
</CsInstruments>
<CsScore>
i1 0 0.1
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*maxarray*, *minarray*, *sumarray*,

## Credits

Author: John ffitch

October 2011

New in Csound version 5.14

Renamed in Csound version 6.00

# scanhammer

scanhammer — Copies from one table to another with a gain control.

## Description

This is a variant of *tablecopy*, copying from one table to another, starting at *ipos*, and with a gain control. The number of points copied is determined by the length of the source. Other points are not changed. This opcode can be used to “hit” a string in the scanned synthesis code.

## Syntax

```
scanhammer isrc, idst, ipos, imult
```

## Initialization

*isrc* -- source function table.

*idst* -- destination function table.

*ipos* -- starting position (in points).

*imult* -- gain multiplier. A value of 0 will leave values unchanged.

## See Also

*scantable*

## Credits

Author: Matt Gilliard  
April 2002

New in version 4.20

# scans

scans — Generate audio output using scanned synthesis.

## Description

Generate audio output using scanned synthesis.

## Syntax

```
ares scans kamp, kfreq, ifn, id [, iorder]
```

## Initialization

*ifn* -- ftable containing the scanning trajectory. This is a series of numbers that contains addresses of masses. The order of these addresses is used as the scan path. It should not contain values greater than the number of masses, or negative numbers. See the *introduction to the scanned synthesis section*.

*id* -- ID number of the *scanu* opcode's waveform to use

*iorder* (optional, default=0) -- order of interpolation used internally. It can take any value in the range 1 to 4, and defaults to 4 if outside that range, which is quartic interpolation. The setting of 3 is cubic, 2 is quadratic and 1 is linear. The higher numbers are slower, but not necessarily better.

## Performance

*kamp* -- output amplitude. Note that the resulting amplitude is also dependent on instantaneous value in the wavetable. This number is effectively the scaling factor of the wavetable.

*kfreq* -- frequency of the scan rate

## Examples

Here is an example of the scanned synthesis. It uses the file *scans.csd* [examples/scans.csd], and *string-128.matrix* [examples/string-128.matrix].

### Example 849. Example of the scans opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o scans.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 128
```

```

    nchnls = 1

    instr 1
a0 = 0
; scanu init, irate, ifnvel, ifnmass, ifnstif, ifncentr, ifndamp, kmass, kstif, kcentr, kdamp, ileft
scanu 1, .01, 6, 2, 3, 4, 5, 2, .1, .1, -.01, .1,
;ar scans kamp, kfreq, ifntraj, id
a1 scans ampdb(p4), cpspch(p5), 7, 2
out a1
endin

</CsInstruments>
<CsScore>

; Initial condition
f1 0 128 7 0 64 1 64 0

; Masses
f2 0 128 -7 1 128 1

; Spring matrices
f3 0 16384 -23 "string-128.matrix"

; Centering force
f4 0 128 -7 0 128 2

; Damping
f5 0 128 -7 1 128 1

; Initial velocity
f6 0 128 -7 0 128 0

; Trajectories
f7 0 128 -5 .001 128 128

; Note list
i1 0 10 86 6.00
i1 11 14 86 7.00
i1 15 20 86 5.00
e

</CsScore>
</CsoundSynthesizer>

```

Here is another example of the scanned synthesis, using samples as excitation signal. It uses the file *scans-2.csd* [examples/scans-2.csd].

### Example 850. Second example of the scans opcode.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o scans-2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

```

```

strset 1, "mary.wav"
strset 2, "fox.wav"

instr 2 ;show 2 different trajectories, with samples as excitation signal

ismp = p6
iamp = p7
itrj = p8
aout soundin p6 ;choose wave file
      scanu ismp, .01, 6, 2, 33, 44, 5, 2, .01, .05, -.05, .1, .5, 0, 0, aout, 1, 0
asig scans iamp, cpspch(p5), itrj, 0
      outs asig, asig

endin
</CsInstruments>
<CsScore>
f1 0 128 7 0 64 1 64 0 ; Initial condition
f2 0 128 -7 1 128 0.3 ; Masses
f33 0 16384 -23 "cylinder-128,8" ; Spring matrices
f44 0 128 -7 2 4 0 124 2 ; Centering force
f5 0 128 -7 1 128 0 ; Damping
f6 0 128 -7 -.0 128 0 ; Initial velocity
f7 0 128 -5 .001 128 128 ; Trajectories
f77 0 128 -23 "spiral-8,16,128,2,lover2"

s
i2 0 5 63 6.00 1 .9 7 ;"mary.wav" &
i2 6 5 60 7.00 ;trajectory table 7
i2 10 5 60 8.00

s
i2 0 5 63 6.00 2 .08 7 ;"fox.wav", at much lower volume
i2 6 5 60 7.00
i2 10 5 60 8.00

s
i2 0 5 63 6.00 1 .9 77 ;"mary.wav" &
i2 6 5 60 7.00 ;trajectory table 77
i2 10 5 60 8.00

s
i2 0 5 63 6.00 2 .08 77 ;"fox.wav", at much lower volume
i2 6 5 60 7.00
i2 10 5 60 8.00
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

The matrix file “string-128.matrix”, as well as several other matrices, is also available in a *zipped file* [<http://www.csounds.com/scanned/zip/scanmatrices.zip>] from the *Scanned Synthesis page* [<http://www.csounds.com/scanned/>] at cSounds.com.

More information on this opcode: [http://www.csounds.com/stevenyi/scanned/yi\\_scannedSynthesis.html](http://www.csounds.com/stevenyi/scanned/yi_scannedSynthesis.html), written by Steven Yi

## Credits

Author: Paris Smaragdis  
MIT Media Lab  
Boston, Massachusetts USA

New in Csound version 4.05

# scantable

scantable — A simpler scanned synthesis implementation.

## Description

A simpler scanned synthesis implementation. This is an implementation of a circular string scanned using external tables. This opcode will allow direct modification and reading of values with the table opcodes.

## Syntax

```
aout scantable kamp, kpch, ipos, imass, istiff, idamp, ivel
```

## Initialization

*ipos* -- table containing position array.

*imass* -- table containing the mass of the string.

*istiff* -- table containing the stiffness of the string.

*idamp* -- table containing the damping factors of the string.

*ivel* -- table containing the velocities.

## Performance

*kamp* -- amplitude (gain) of the string.

*kpch* -- the string's scanned frequency.

## Examples

Here is an example of the scantable opcode. It uses the file *scantable.csd* [examples/scantable.csd].

### Example 851. Example of the scantable opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac --rtmidi=virtual -M0 ;;realtime audio out and midi in
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o scantable.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
```

```
Odbfs = 1

gi1 ftgen 1, 0, 128, 7, 0, 64, 1, 64, 0 ; initial position
gi2 ftgen 2, 0, 128, -7, 1, 128, 1 ; masses
gi3 ftgen 3, 0, 128, -7, 0, 64, 100, 64, 0 ; stiffness
gi4 ftgen 4, 0, 128, -7, 1, 128, 1 ; damping
gi5 ftgen 5, 0, 128, -7, 0, 128, 0.5 ; initial velocity

instr 1

iamp ampmidi .5
ipch cpsmidi
kenv madsr .1, .1, .8, .3

asig scantable iamp, ipch, 1, 2, 3, 4, 5
asig dcblock asig
    outs asig*kenv, asig*kenv

endin
</CsInstruments>
<CsScore>

f0 60 ; play for 60 seconds
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*scanhammer*

More information on this opcode: [http://www.csounds.com/stevenyi/scanned/yi\\_scannedSynthesis.html](http://www.csounds.com/stevenyi/scanned/yi_scannedSynthesis.html) ,  
written by Steven Yi

## Credits

Author: Matt Gilliard  
April 2002

New in version 4.20

## scanu

scanu — Compute the waveform and the wavetable for use in scanned synthesis.

## Description

Compute the waveform and the wavetable for use in scanned synthesis.

## Syntax

```
scanu init, irate, ifnvel, ifnmass, ifnstif, ifncentr, ifndamp, kmass, \  
      kstif, kcentr, kdamp, ileft,  iright, kpos, kstrngth, ain, idisp, id
```

## Initialization

*init* -- the initial position of the masses. If this is a negative number, then the absolute of *init* signifies the table to use as a hammer shape. If *init* > 0, the length of it should be the same as the intended mass number, otherwise it can be anything.

*ifnvel* -- the ftable that contains the initial velocity for each mass. It should have the same size as the intended mass number.

*ifnmass* -- ftable that contains the mass of each mass. It should have the same size as the intended mass number.

*ifnstif* -- ftable that contains the spring stiffness of each connection. It should have the same size as the square of the intended mass number. The data ordering is a row after row dump of the connection matrix of the system.

*ifncentr* -- ftable that contains the centering force of each mass. It should have the same size as the intended mass number.

*ifndamp* -- the ftable that contains the damping factor of each mass. It should have the same size as the intended mass number.

*ileft* -- If *init* < 0, the position of the left hammer (*ileft* = 0 is hit at leftmost, *ileft* = 1 is hit at rightmost).

*iright* -- If *init* < 0, the position of the right hammer (*iright* = 0 is hit at leftmost, *iright* = 1 is hit at rightmost).

*idisp* -- If 0, no display of the masses is provided.

*id* -- If positive, the ID of the opcode. This will be used to point the scanning opcode to the proper waveform maker. If this value is negative, the absolute of this value is the wavetable on which to write the waveshape. That wavetable can be used later from an other opcode to generate sound. The initial contents of this table will be destroyed.

## Performance

*kmass* -- scales the masses

*kstif* -- scales the spring stiffness

*kcentr* -- scales the centering force

*kdamp* -- scales the damping

*kpos* -- position of an active hammer along the string (*kpos* = 0 is leftmost, *kpos* = 1 is rightmost). The shape of the hammer is determined by *init* and the power it pushes with is *kstrngth*.

*kstrngth* -- power that the active hammer uses

*ain* -- audio input that adds to the velocity of the masses. Amplitude should not be too great.

## Examples

For an example, see the documentation on *scans*.

## See Also

More information on this opcode: [http://www.csounds.com/stevenyi/scanned/yi\\_scannedSynthesis.html](http://www.csounds.com/stevenyi/scanned/yi_scannedSynthesis.html) , written by Steven Yi

## Credits

Author: Paris Smaragdis  
MIT Media Lab  
Boston, Massachusetts USA  
March 2000

New in Csound version 4.05

# schedkwhen

`schedkwhen` — Adds a new score event generated by a k-rate trigger.

## Description

Adds a new score event generated by a k-rate trigger.

## Syntax

```
schedkwhen ktrigger, kmintim, kmaxnum, kinsnum, kwhen, kdur \
  [, ip4] [, ip5] [...]
```

```
schedkwhen ktrigger, kmintim, kmaxnum, "insname", kwhen, kdur \
  [, ip4] [, ip5] [...]
```

## Initialization

*“insname”* -- A string (in double-quotes) representing a named instrument.

*ip4, ip5, ...* -- Equivalent to *p4, p5, etc.*, in a score *i statement*

## Performance

*ktrigger* -- triggers a new score event. If *ktrigger* = 0, no new event is triggered.

*kmintim* -- minimum time between generated events, in seconds. If *kmintim* <= 0, no time limit exists. If the *kinsnum* is negative (to turn off an instrument), this test is bypassed.

*kmaxnum* -- maximum number of simultaneous instances of instrument *kinsnum* allowed. If the number of existant instances of *kinsnum* is >= *kmaxnum*, no new event is generated. If *kmaxnum* is <= 0, it is not used to limit event generation. If the *kinsnum* is negative (to turn off an instrument), this test is bypassed.

*kinsnum* -- instrument number. Equivalent to *p1* in a score *i statement*.

*kwhen* -- start time of the new event. Equivalent to *p2* in a score *i statement*. Measured from the time of the triggering event. *kwhen* must be >= 0. If *kwhen* > 0, the instrument will not be initialized until the actual time when it should start performing.

*kdur* -- duration of event. Equivalent to *p3* in a score *i statement*. If *kdur* = 0, the instrument will only do an initialization pass, with no performance. If *kdur* is negative, a held note is initiated. (See *ihold* and *i statement*.)



### Note

While waiting for events to be triggered by `schedkwhen`, the performance must be kept going, or Csound may quit if no score events are expected. To guarantee continued performance, an *f0 statement* may be used in the score.



### Note

Note that the `schedkwhen` opcode can't accept string p-fields. If you need to pass strings when instantiating an instrument, use the `scoreline` or `scoreline_i` opcode.

## Examples

Here is an example of the `schedkwhen` opcode. It uses the file `schedkwhen.csd` [examples/schedkwhen.csd].

### Example 852. Example of the `schedkwhen` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc    ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o schedkwhen.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1 - oscillator with a high note.
instr 1
; Use the fourth p-field as the trigger.
ktrigger = p4
kmintim = 0
kmaxnum = 2
kinsnum = 2
kwhen = 0
kdur = 0.5

; Play Instrument #2 at the same time, if the trigger is set.
schedkwhen ktrigger, kmintim, kmaxnum, kinsnum, kwhen, kdur

; Play a high note.
a1 oscils 10000, 880, 1
out a1
endin

; Instrument #2 - oscillator with a low note.
instr 2
; Play a low note.
a1 oscils 10000, 220, 1
out a1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; p4 = trigger for Instrument #2 (when p4 > 0).
; Play Instrument #1 for half a second, no trigger.
i 1 0 0.5 0
; Play Instrument #1 for half a second, trigger Instrument #2.
i 1 1 0.5 1
e

```

```
</CsScore>  
</CsoundSynthesizer>
```

## See also

*event, event\_i, schedule, schedwhen, schedkwhennamed, scoreline, scoreline\_i*

## Credits

Author: Rasmus Ekman  
EMS, Stockholm, Sweden

Example written by Kevin Conder.

New in Csound version 3.59

# schedkwhennamed

`schedkwhennamed` — Similar to `schedkwhen` but uses a named instrument at init-time.

## Description

Similar to *schedkwhen* but uses a named instrument at init-time.

## Syntax

```
schedkwhennamed ktrigger, kmintim, kmaxnum, "name", kwhen, kdur \
  [, ip4] [, ip5] [...]
```

## Initialization

*ip4*, *ip5*, ... -- Equivalent to *p4*, *p5*, etc., in a score *i statement*

## Performance

*ktrigger* -- triggers a new score event. If *ktrigger* is 0, no new event is triggered.

*kmintim* -- minimum time between generated events, in seconds. If *kmintim* is less than or equal to 0, no time limit exists.

*kmaxnum* -- maximum number of simultaneous instances of named instrument allowed. If the number of existant instances of the named instrument is greater than or equal to *kmaxnum*, no new event is generated. If *kmaxnum* is less than or equal to 0, it is not used to limit event generation.

"*name*" -- the named instrument's name.

*kwhen* -- start time of the new event. Equivalent to *p2* in a score *i statement*. Measured from the time of the triggering event. *kwhen* must be greater than or equal to 0. If *kwhen* greater than 0, the instrument will not be initialized until the actual time when it should start performing.

*kdur* -- duration of event. Equivalent to *p3* in a score *i statement*. If *kdur* is 0, the instrument will only do an initialization pass, with no performance. If *kdur* is negative, a held note is initiated. (See *ihold* and *i statement*.)



### Note

While waiting for events to be triggered by *schedkwhennamed*, the performance must be kept going, or Csound may quit if no score events are expected. To guarantee continued performance, an *f0 statement* may be used in the score.



### Note

Note that the *schedkwhennamed* opcode can't accept string p-fields. If you need to pass strings when instantiating an instrument, use the *scoreline* or *scoreline\_i* opcode.

## Examples

Here is an example of the *schedkwhennamed* opcode. It uses the file *schedkwhennamed.csd* [examples/schedkwhennamed.csd].

**Example 853. Example of the schedkwhennamed opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d
; For Non-realtime ouput leave only the line below:
; -o schedkwhennamed.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

  sr      = 48000
  ksmps   = 16
  nchnls  = 2
  0dbfs   = 1

; Example by Jonathan Murphy 2007

gSinstr2 = "printer"

  instr 1

  ktrig   metro      1
  if (ktrig == 1) then
    ;Call instrument "printer" once per second
    schedkwhennamed ktrig, 0, 1, gSinstr2, 0, 1
  endif

  endin

  instr printer

  ktime   timeinsts
  printk2 ktime

  endin

</CsInstruments>
<CsScore>
i1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

**See also**

*event, event\_i, schedule, schedwhen, schedkwhen, scoreline, scoreline\_i*

**Credits**

Author: Rasmus Ekman  
EMS, Stockholm, Sweden

New in Csound version 4.23

# schedule

schedule — Adds a new score event.

## Description

Adds a new score event.

## Syntax

```
schedule insnum, iwhen, idur [, ip4] [, ip5] [...]
```

```
schedule "insname", iwhen, idur [, ip4] [, ip5] [...]
```

## Initialization

*insnum* -- instrument number. Equivalent to p1 in a score *i statement*. *insnum* must be a number greater than the number of the calling instrument.

*"insname"* -- A string (in double-quotes) representing a named instrument.

*iwhen* -- start time of the new event. Equivalent to p2 in a score *i statement*. *iwhen* must be nonnegative. If *iwhen* is zero, *insnum* must be greater than or equal to the p1 of the current instrument.

*idur* -- duration of event. Equivalent to p3 in a score *i statement*. .

*ip4, ip5, ...* -- Equivalent to p4, p5, etc., in a score *i statement*. The opcode also accepts strings as arguments for p4-pN.

## Performance

*schedule* adds a new score event. The arguments, including options, are the same as in a score. The *iwhen* time (p2) is measured from the time of this event.

If the duration is zero or negative the new event is of MIDI type, and inherits the release sub-event from the scheduling instruction.

## Examples

Here is an example of the *schedule* opcode. It uses the file *schedule.csd* [examples/schedule.csd].

### Example 854. Example of the *schedule* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
```

```
; For Non-realtime ouput leave only the line below:
; -o schedule.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - oscillator with a high note.
instr 1
; Play Instrument #2 at the same time.
schedule 2, 0, p3

; Play a high note.
a1 oscils 10000, 880, 1
out a1
endin

; Instrument #2 - oscillator with a low note.
instr 2
; Play a low note.
a1 oscils 10000, 220, 1
out a1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for half a second.
i 1 0 0.5
; Play Instrument #1 for half a second.
i 1 1 0.5
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*schedwhen*

## See also

*event, event\_i, schedwhen, schedkwhen, schedkwhennamed, scoreline, scoreline\_i*

More information on this opcode: [http://www.csoundjournal.com/issue15/phrase\\_loops.html](http://www.csoundjournal.com/issue15/phrase_loops.html) , written by Jim Aikin

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
November 1998

Example written by Kevin Conder.

New in Csound version 3.491

Based on work by Gabriel Maldonado

Thanks goes to David Gladstein, for clarifying the *when* parameter.

# schedwhen

schedwhen — Adds a new score event.

## Description

Adds a new score event.

## Syntax

```
schedwhen ktrigger, kinsnum, kwhen, kdur [, ip4] [, ip5] [...]
```

```
schedwhen ktrigger, "insname", kwhen, kdur [, ip4] [, ip5] [...]
```

## Initialization

*ip4*, *ip5*, ... -- Equivalent to *p4*, *p5*, etc., in a score *i statement*.

## Performance

*kinsnum* -- instrument number. Equivalent to *p1* in a score *i statement*.

*"insname"* -- A string (in double-quotes) representing a named instrument.

*ktrigger* -- trigger value for new event

*kwhen* -- start time of the new event. Equivalent to *p2* in a score *i statement*.

*kdur* -- duration of event. Equivalent to *p3* in a score *i statement*.

*schedwhen* adds a new score event. The event is only scheduled when the k-rate value *ktrigger* is first non-zero. The arguments, including options, are the same as in a score. The *kwhen* time (*p2*) is measured from the time of this event.

If the duration is zero or negative the new event is of MIDI type, and inherits the release sub-event from the scheduling instruction.



### Note

Note that the *schedwhen* opcode can't accept string *p*-fields. If you need to pass strings when instantiating an instrument, use the *scoreline* or *scoreline\_i* opcode.

## Examples

Here is an example of the *schedwhen* opcode. It uses the file *schedwhen.csd* [examples/schedwhen.csd].

### Example 855. Example of the schedwhen opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o schedwhen.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kmtr metro 100      ;produce 100 triggers per second
      schedwhen kmtr, 2, 1, .3 ;but schedwhen plays instr. 2 only once

endin

instr 2

aenv linseg 0, p3*.1, 1, p3*.3, 1, p3*.6, 0 ;envelope
a1  poscil .3*aenv, 1000, 1
     outs a1, a1

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine

i 1 0 3
i 1 3 5
e
</CsScore>
</CsoundSynthesizer>
```

## See also

*event, event\_i, schedule, schedkwhen, schedkwhennamed, scoreline, scoreline\_i*

## Credits

Author: John ffitth  
University of Bath/Codemist Ltd.  
Bath, UK  
November 1998

New in Csound version 3.491

Based on work by Gabriel Maldonado

# scoreline

scoreline — Issues one or more score line events from an instrument.

## Description

Scoreline will issue one or more score events, if *ktrig* is 1 every *k*-period. It can handle strings in the same conditions as the standard score. Multi-line strings are accepted, using `{{ }}` to enclose the string.

## Syntax

```
scoreline Sin, ktrig
```

## Initialization

“*Sin*” -- a string (in double-quotes or enclosed by `{{ }}`) containing one or more score events.

## Performance

“*ktrig*” -- event trigger, 1 issues the score event, 0 bypasses it.

## Examples

Here is an example of the scoreline opcode. It uses the file *scoreline.csd* [examples/scoreline.csd].

### Example 856. Example of the scoreline opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o scoreline.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ktrig metro 3      ;trigger 3 times a second
scoreline {{      ;so it sounds like an echo
    i 2 0 3 "flute.aiff"
    i 2 1 3 "beats.wav"
}}, ktrig

ktrig = 0
```

```
    endin
instr 2
asig soundin p4
    outs asig*.3, asig*.3

    endin
</CsInstruments>
<CsScore>

i1 0 2 ;play for 2 seconds, so the samples are played 6 times
e
</CsScore>
</CsoundSynthesizer>
```

You can use string opcodes like *sprintfk* to produce strings to be passed to *scoreline* like this:

```
Sfil = "/Volumes/Bla/file.aif"
String sprintfk {{i 2 0 %f "%s" %f %f %f %f}}, idur, Sfil, p5, p6, knorm, iskip
scoreline String, ktrig
```

## See also

*event*, *event\_i*, *schedule*, *schedwhen*, *schedkwhen*, *schedkwhennamed*, *scoreline\_i*

More information on this opcode: [http://www.csoundjournal.com/issue15/phrase\\_loops.html](http://www.csoundjournal.com/issue15/phrase_loops.html) , written by Jim Aikin

And in the Floss Manuals: [http://en.flossmanuals.net/csound/ch020\\_e-triggering-instrument-events/](http://en.flossmanuals.net/csound/ch020_e-triggering-instrument-events/).

## Credits

Author: Victor Lazzarini, 2007

# scoreline\_i

scoreline\_i — Issues one or more score line events from an instrument at i-time.

## Description

scoreline\_i will issue score events at i-time. It can handle strings in the same conditions as the standard score. Multi-line strings are accepted, using {{ }} to enclose the string.

## Syntax

```
scoreline_i Sin
```

## Initialization

“Sin” -- a string (in double-quotes or enclosed by {{ }}) containing one or more score events.

## Examples

Here is an example of the scoreline\_i opcode. It uses the file *scoreline\_i.csd* [examples/scoreline\_i.csd].

### Example 857. Example of the scoreline\_i opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac --old-parser   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o scoreline.wav -W   ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

scoreline_i {{
    i 2 0 3 "flute.aiff"
    i 2 1 3 "beats.wav"
}}

endin

instr 2

asig soundin p4
outs asig*.8, asig*.8

endin
```

```
</CsInstruments>
<CsScore>

i1 0 1
e
</CsScore>
</CsoundSynthesizer>
```

## See also

*event, event\_i, schedule, schedwhen, schedkwhen, schedkwhennamed, scoreline*

More information on this opcode: [http://www.csoundjournal.com/issue15/phrase\\_loops.html](http://www.csoundjournal.com/issue15/phrase_loops.html) , written by Jim Aikin

And in the Floss Manuals: [http://en.flossmanuals.net/csound/ch020\\_e-triggering-instrument-events/](http://en.flossmanuals.net/csound/ch020_e-triggering-instrument-events/).

## Credits

Author: Victor Lazzarini, 2007

# sc\_lag

sc\_lag — Exponential Lag

## Description

Exponential lag with 60dB lag time. Port of Supercollider's Lag

## Syntax

```
kout sc_lag ain, klagtime [, initialvalue=0]
kout sc_lag kin, klagtime [, initialvalue=0]
```

## Initialization

*initialvalue* If given, sets the internal state. Default=0

## Performance

*ain* -- input signal

*klagtime* -- 60 dB lag time in seconds.

*kladown* -- 60 dB lag time in seconds for the downgoing signal.

This is essentially the same as one pole except that instead of supplying the coefficient directly, it is calculated from a 60 dB lag time. This is the time required for the filter to converge to within 0.01% of a value. This is useful for smoothing out control signals.

## Examples

Here is an example of the sc\_lag opcode. It uses the file *sc\_lag.csd* [examples/sc\_lag.csd].

### Example 858. Example of the sc\_lag opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 128
nchnls = 2
0dbfs = 1.0

instr 1
; smooth a krate signal
kx = floor(line(0, p3, 10))
kx2 = sc_lag(kx, 0.1)
printk2 kx2
```

```
endin

instr 2
; smooth an audio signal
kmidi = floor(line(60, p3, 72)/2)*2
afreq = upsamp(mtof(kmidi))
afreqsmooth = sc_lag(afreq, 1)
a1 = oscili(1, afreq)
a2 = oscili(1, afreqsmooth)
outch 1, a1
outch 2, a2
endin

</CsInstruments>
<CsScore>
i 1 0 5
i 2 0 10

</CsScore>
</CsoundSynthesizer>
```

## See Also

*sc\_lagud, port, portk*

## Credits

By: Eduardo Moguillansky 2017

# sc\_lagud

sc\_lagud — Exponential Lag

## Description

Exponential lag with different smoothing time for up- and downgoing signals. Port of Supercollider's LagUD

## Syntax

```
aout sc_lagud ain, klagup, klagdown
```

```
kout sc_lagud kin, klagup, klagdown
```

## Initialization

*initialvalue* If given, sets the internal state. Default=0

## Performance

*ain* -- input signal

*klagup* -- 60 dB lag time in seconds for the upgoing signal.

*klagdown* -- 60 dB lag time in seconds for the downgoing signal.

This is essentially the same as `sc_lag` except that you can supply a different 60 dB time for when the signal goes up, from when the signal goes down. This is useful for smoothing out control signals, where "fade in" should be different from "fade out".

## Examples

Here is an example of the `sc_lagud` opcode. It uses the file `sc_lagud.csd` [examples/sc\_lagud.csd].

### Example 859. Example of the `sc_lagud` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 128
nchnls = 2
0dbfs = 1.0

instr 1
  kx = randh(2, 1)
  kx2 = sc_lagud(kx, 1.0, 0.1, 1)
```

```
    printks "x %f x2 %f\n", 0.1, kx, kx2
endin

instr 2
  kmidis[] array 60, 65, 60, 65, 60
  ilen = lenarray(kmidis)
  kidx = int(linseg(0, ilen*2, ilen-0.00000001))
  kmidi = mtof(kmidis[kidx])
  afreq = sc_lagud(a(kmidi), 1, 0.1)
  a0 = oscili(0.7, afreq)
  kfreq = sc_lagud(kmidi, 1, 0.1)
  a1 = oscili(0.7, kfreq)
  outch 1, a0
  outch 2, a1
endin

</CsInstruments>
<CsScore>
; i 1 0 10
i 2 0 12

</CsScore>
</CsoundSynthesizer>
```

## See Also

*sc\_lag, port, portk*

## Credits

By: Eduardo Moguillansky 2017

# sc\_phasor

sc\_phasor — A resettable linear ramp between two levels

## Description

A resettable linear ramp between two levels. Port of Supercollider's Phasor.

## Syntax

```
aindex sc_phasor xtrig, xrate, kstart, kend [, kresetPos]
```

```
kindex sc_phasor xtrig, xrate, kstart, kend [, kresetPos]
```

## Performance

Phasor is a linear ramp between *kstart* and *kend* values. When its trigger input crosses from non-positive to positive, *sc\_phasor*'s output will jump to its reset position (or to *kstart* if no reset value was given). Upon reaching the end of its ramp, *sc\_phasor* will wrap back to its start.

If *kresetPos* is specified, this value will be used to jump to after a trigger. Otherwise, a trigger will set the position back to *kstart*

If one wants Phasor to output a signal with frequency *freq* oscillating between *start* and *end*, then the *rate* should be:

```
(end - start) * freq / sr
```

## Examples

Here is an example of the *sc\_phasor* opcode. It uses the file *sc\_phasor.csd* [examples/sc\_phasor.csd].

### Example 860. Example of the *sc\_phasor* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 128
nchnls = 2
0dbfs = 1.0

instr 1
  krate linseg 1, p3, 40
  ktrig metro krate
  kx sc_phasor ktrig, krate/kr, 0, 1
  asine oscili 0.2, kx*500+500
  outch 1, asine
endin
```

```
instr 2
  krate linseg 1, p3, 40
  atrig = mpulse(1, 1/krate)
  ax sc phasor atrig, krate/sr, 0, 1
  asine oscili 0.2, ax*500+500
  outch 2, asine
endin

</CsInstruments>
<CsScore>
i1 0 20
i2 0 20
</CsScore>
</CsoundSynthesizer>
```

## See also

*phasor*

## Credits

By: Eduardo Moguillansky 2017

# sc\_trig

sc\_trig — Timed trigger

## Description

Timed trigger. Port of Supercollider's Trig ugen

## Syntax

```
aout sc_trig ain, kdur
```

```
kout sc_trig kin, kdur
```

## Initialization

*initialvalue* If given, sets the internal state. Default=0

## Performance

*ain* -- Trigger. Trigger can be any signal. A trigger happens when the signal changes from non-positive to positive.

*kdur* -- Duration of the trigger in seconds.

When a nonpositive to positive transition occurs at the input, Trig outputs the level of the triggering input for the specified duration, otherwise it outputs zero.

## Examples

Here is an example of the sc\_trig opcode. It uses the file *sc\_trig.csd* [examples/sc\_trig.csd].

### Example 861. Example of the sc\_trig opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 128
nchnls = 2
0dbfs = 1.0

instr 1
  km = metro(1)
  kt timeinstd
  ktrig = sc_trig(km, 0.5)
  printks "t=%f km=%f ktrig=%f\n", 0.01, kt, km, ktrig
endin
```

```
instr 2
  am = upsamp(metro(1))
  aenv = sc_trig(am, 0.5)
  asig pinker
  outch 1, asig*aenv
  outch 2, asig
endin

</CsInstruments>
<CsScore>
i 1 0 10
i 2 0 10

</CsScore>
</CsoundSynthesizer>
```

## See Also

*sc\_lag, metro, mpulse trigger*

## Credits

By: Eduardo Moguillansky 2017

# seed

seed — Sets the global seed value.

## Description

Sets the global seed value for all *x-class noise generators*, as well as other opcodes that use a random call, such as *grain*.



### Please Note

*rand*, *randh*, *randi*, *rnd(x)* and *birnd(x)* are not affected by seed.

## Syntax

```
seed ival
```

## Performance

Use of *seed* will provide predictable results from an orchestra using with random generators, when required from multiple performances.

When specifying a seed value, *ival* should be an integer between 0 and  $2^{32}$ . If *ival* = 0, the value of *ival* will be derived from the system clock.

## Examples

Here is an example of the seed opcode. It uses the file *seed.csd* [examples/seed.csd].

### Example 862. Example of the seed opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o seed.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;same values every time

seed 10
krnd randomh 100, 200, 5
printk .5, krnd ; look
```

```
aout oscili 0.8, 440+krnd, 1 ; & listen
outs aout, aout

endin

instr 2 ;different values every time - value is derived from system clock

seed 0 ; seed from system clock
krnd randomh 100, 200, 5
printk .5, krnd ; look
aout oscili 0.8, 440+krnd, 1 ; & listen
outs aout, aout

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine wave.

i 1 0 1
i 2 2 1
e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like:

```
i 1 time 0.00067: 100.00000
i 1 time 0.50000: 175.78677
i 1 time 1.00000: 170.89579
```

WARNING: Seeding from current time 834128659

```
i 2 time 2.00067: 100.00000
i 2 time 2.50000: 197.58517
i 2 time 3.00000: 188.69525
```

# sekere

sekere — Semi-physical model of a sekere sound.

## Description

*sekere* is a semi-physical model of a sekere sound. It is one of the PhISEM percussion opcodes. PhISEM (Physically Informed Stochastic Event Modeling) is an algorithmic approach for simulating collisions of multiple independent sound producing objects.

## Syntax

```
ares sekere iamp, idettack [, inum] [, idamp] [, imaxshake]
```

## Initialization

*iamp* -- Amplitude of output. Note: As these instruments are stochastic, this is only a approximation.

*idettack* -- period of time over which all sound is stopped

*inum* (optional) -- The number of beads, teeth, bells, timbrels, etc. If zero, the default value is 64.

*idamp* (optional) -- the damping factor, as part of this equation:

$$\text{damping\_amount} = 0.998 + (\text{idamp} * 0.002)$$

The default *damping\_amount* is 0.999 which means that the default value of *idamp* is 0.5. The maximum *damping\_amount* is 1.0 (no damping). This means the maximum value for *idamp* is 1.0.

The recommended range for *idamp* is usually below 75% of the maximum value.

*imaxshake* (optional) -- amount of energy to add back into the system. The value should be in range 0 to 1.

## Examples

Here is an example of the sekere opcode. It uses the file *sekere.csd* [examples/sekere.csd].

### Example 863. Example of the sekere opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o sekere.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
```

```
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

idamp = p4 ;vary damping amount
asig sekere 1, 0.01, 64, idamp
outs asig, asig

endin
</CsInstruments>
<CsScore>

i1 0 1 .1
i1 + 1 .9
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*cabasa, crunch, sandpaper, stix*

## Credits

Author: Perry Cook, part of the PhISEM (Physically Informed Stochastic Event Modeling)

Adapted by John ffitch

University of Bath, Codemist Ltd.

Bath, UK

New in Csound version 4.07

Added notes by Rasmus Ekman on May 2002.

# select

select — Select sample value based on audio-rate comparisons.

## Description

Select sample value from three based on audio-rate comparisons of two signals.

## Syntax

```
aout select a1, a2, aless, aequal, amore
```

## Performance

*a1*, *a2* -- audio signals that are compared.

*aless* -- audio signal selected if  $a1[n] < a2[n]$

*aequal* -- audio signal selected if  $a1[n] = a2[n]$

*asig* -- audio signal selected if  $a1[n] > a2[n]$

The comparison is made sample by sample.

## Examples

Here is an example of the select opcode. It uses the file *select.csd* [examples/select.csd].

### Example 864. Example of the select opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>

<CsInstruments>
nchnls = 3
0dbfs = 1

instr 1
  a1 oscil 0.8, 440
  a2 oscil 0.6, 880
  az = 0
  alow line -1, p3, -1
  ahigh line 1, p3, 1
  aout select a1,a2,alow,az,ahigh
  outch 1, a1, 2,a2, 3,aout
endin
</CsInstruments>

<CsScore>
i 1 0 5
e
</CsScore>
```

`</CsoundSynthesizer>`

## See Also

*deltap*

## Credits

By: John ffitich 2016

New in vesion 6.09

# semitone

semitone — Calculates a factor to raise/lower a frequency by a given amount of semitones.

## Description

Calculates a factor to raise/lower a frequency by a given amount of semitones.

## Syntax

```
semitone(x)
```

This function works at a-rate, i-rate, and k-rate.

## Initialization

*x* -- a value expressed in semitones.

## Performance

The value returned by the *semitone* function is a factor. You can multiply a frequency by this factor to raise/lower it by the given amount of semitones.

## Examples

Here is an example of the semitone opcode. It uses the file *semitone.csd* [examples/semitone.csd].

### Example 865. Example of the semitone opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o semitone.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

iroot = 440 ; root note is A above middle-C (440 Hz)
ksem lfo 12, .5, 5 ; generate sawtooth, go from 5 octaves higher to root
ksm = int(ksem) ; produce only whole numbers
kfactor = semitone(ksm) ; for semitones
knew = iroot * kfactor
```

```
printk2 knew
printk2 kfactor
asig pluck 1, knew, 1000, 0, 1
asig dcblock asig ;remove DC
outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 5
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like:

```
i1 880.00000
i1 2.00000
i1 830.65625
i1 1.88786
i1 783.94665
i1 1.78170
i1 739.98885
i1 1.68179
i1 698.49586
i1 1.58749
i1 659.21793
i1 1.49822
i1 622.25397
i1 1.41421
i1 587.36267
i1 1.33492
i1 554.33399
i1 1.25985
i1 523.25113
i1 1.18921
i1 493.91116
i1 1.12253
i1 466.13747
i1 1.05940
i1 440.00000
i1 1.00000
.....
```

## See Also

*cent, db, octave*

## Credits

New in version 4.16

## **sense**

sense — Same as the sensekey opcode.

## **Description**

Same as the *sensekey* opcode.

# sensekey

sensekey — Returns the ASCII code of a key that has been pressed.

## Description

Returns the ASCII code of a key that has been pressed, or -1 if no key has been pressed.

## Syntax

```
kres[, kkeydown] sensekey
```

## Performance

*kres* - returns the ASCII value of a key which is pressed or released.

*kkeydown* - returns 1 if the key was pressed, 0 if it was released or if there is no key event.

*kres* can be used to read keyboard events from stdin and returns the ASCII value of any key that is pressed or released, or it returns -1 when there is no keyboard activity. The value of *kkeydown* is 1 when a key was pressed, or 0 otherwise. This behavior is emulated by default, so a key release is generated immediately after every key press. To have full functionality, FLTK can be used to capture keyboard events. *FLpanel* can be used to capture keyboard events and send them to the sensekey opcode, by adding an additional optional argument. See *FLpanel* for more information.



### Note

This opcode can also be written as *sense*.

## Examples

Here is an example of the sensekey opcode. It uses the file *sensekey.csd* [examples/sensekey.csd].

### Example 866. Example of the sensekey opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o sensekey.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1
```

```

; Instrument #1.
instr 1
  k1 sensekey
  printk2 k1
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for thirty seconds.
i 1 0 30
e

</CsScore>
</CsoundSynthesizer>

```

Here is what the output should look like when the "q" button is pressed...

```
q i1 113.00000
```

Here is an example of the sensekey opcode in conjunction with *FLpanel*. It uses the file *FLpanel-sensekey.csd* [examples/FLpanel-sensekey.csd].

### Example 867. Example of the sensekey opcode using keyboard capture from an FLpanel.

```

<CsoundSynthesizer>

<CsOptions>
; Select audio/midi flags here according to platform
; Audio out Audio in No messages
-odac -iadc -d ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o FLpanel-sensekey.wav -W ;; for file output any platform
</CsOptions>

<CsInstruments>
; Example by Johnathan Murphy

sr = 44100
ksmps = 128
nchnls = 2

; ikbdcapture flag set to 1
ikey init 1

FLpanel "sensekey", 740, 340, 100, 250, 2, ikey
gkasc, giasc FLbutBank 2, 16, 8, 700, 300, 20, 20, -1
FLpanelEnd
FLrun

instr 1

kkey sensekey
kprint changed kkey
FLsetVal kprint, kkey, giasc

endin

```

```

</CsInstruments>

<CsScore>
i1 0 60
e
</CsScore>

</CsoundSynthesizer>

```

The lit button in the FLpanel window shows the last key pressed.

Here is a more complex example of the sensekey opcode in conjunction with *FLpanel*. It uses the file *FLpanel-sensekey2.csd* [examples/FLpanel-sensekey.csd].

### Example 868. Example of the sensekey opcode using keyboard capture from an FLpanel.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out Audio in No messages
-odac ; -iadc -d ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o FLpanel-sensekey2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
sr = 48000
ksmps = 32
nchnls = 1
; Example by Istvan Varga
; if the FLTK opcodes are commented out, sensekey will read keyboard
; events from stdin
FLpanel "", 150, 50, 100, 100, 0, 1
FLlabel 18, 10, 1, 0, 0, 0
FLgroup "Keyboard Input", 150, 50, 0, 0, 0
FLgroupEnd
FLpanelEnd

FLrun

instr 1

ktrig1 init 1
ktrig2 init 1
nxtKey1:
k1, k2 sensekey
if (k1 != -1 || k2 != 0) then
printf "Key code = %02X, state = %d\n", ktrig1, k1, k2
ktrig1 = 3 - ktrig1
kgoto nxtKey1
endif
nxtKey2:
k3 sensekey
if (k3 != -1) then
printf "Character = '%c'\n", ktrig2, k3
ktrig2 = 3 - ktrig2
kgoto nxtKey2
endif

endin

</CsInstruments>
<CsScore>

```

```
i 1 0 3600  
e  
</CsScore>  
</CsoundSynthesizer>
```

The console output will look something like:

```
new alloc for instr 1:  
Key code = 65, state = 1  
Character = 'e'  
Key code = 65, state = 0  
Key code = 72, state = 1  
Character = 'r'  
Key code = 72, state = 0  
Key code = 61, state = 1  
Character = 'a'  
Key code = 61, state = 0
```

## See also

*FLpanel*, *FLkeyIn*

## Credits

Author: John ffitch  
University of Bath, Codemist. Ltd.  
Bath, UK  
October 2000

Examples written by Kevin Conder, Johnathan Murphy and Istvan Varga.

New in Csound version 4.09. Renamed in Csound version 4.10.

# serialBegin

serialBegin — Open a serial port.

## Description

Open a serial port for arduino.

## Syntax

```
iPort serialBegin SPortName [, iBaudRate]
```

## Initialization

*SPortName* -- port name number

*iBaudRate* -- serial speed, defaulting to 9600 bps.

## Examples

Here is an example of the serialBegin opcode. It uses the file *serialBegin.csd* [examples/serialBegin.csd].

### Example 869. Example of the serialBegin opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n   ;;no output
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o serialBegin.wav -W   ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 500 ; the default krate can be too fast for the arduino to handle
nchnls = 2
0dbfs = 1

instr 1

iPort serialBegin "/dev/cu.usbmodemfa131", 9600   ;connect to the arduino with baudrate = 9600

kGain init 16           ;read our knob value
kVal serialRead iPort
if (kVal != -1) then
    kGain = kVal/128
endif

aSig in                ;get our audio input and get its rms
kRms rms aSig*kGain

kRms = kRms*kRms*255           ;scale the rms to a good value for the LED and send it out
```

```
serialWrite iPort, (kRms < 255 ? kRms : 255) ;must be in range: 0-255

endin
</CsInstruments>
<CsScore>
f 1 0 1024 10 1 1 1 1 1 1

i 1 0 200
e
</CsScore>
</CsoundSynthesizer>
```

This is the matching Arduino code :

```
void setup() {
  // enable serial communication
  Serial.begin(9600);

  // declare pin 9 to be an output:
  pinMode(9, OUTPUT);
}

void loop() {
  // only do something if we received something (this should be at csound's k-rate)
  if (Serial.available()) {
    // set the brightness of LED (connected to pin 9) to our input value
    int brightness = Serial.read();
    analogWrite(9, brightness);

    // while we are here, get our knob value and send it to csound
    int sensorValue = analogRead(A0);
    Serial.write(sensorValue/4); // scale to 1-byte range (0-255)
  }
}
.....
```

## See Also

[serialEnd](#), [serialWrite\\_i](#), [serialWrite](#), [serialRead](#), [serialPrint](#), [serialFlush](#).

## Credits

Author: Matt Ingalls  
2011

New in version 5.14

# serialEnd

`serialEnd` — Close a serial port.

## Description

Close a serial port for arduino.

## Syntax

```
serialEnd iPort
```

## Initialization

*iPort* -- port number obtained from a *serialBegin*opcode.

## See Also

`serialBegin`, `serialWrite_i`, `serialWrite`, `serialRead`, `serialPrint`, `serialFlush`.

## Credits

Author: Matt Ingalls  
2011

New in version 5.14

# serialFlush

`serialFlush` — Flush data from a serial port.

## Description

Flush to the screen any bytes (up to 32k) in the input buffer. Note that these bytes will be cleared from the buffer. use this opcode mainly for debugging messages. If you want to mix debugging and other communication messages over the same port, you will need to manually parse the data with the *serialRead* opcode.

## Syntax

```
serialFlush iPort
```

## Performance

*iPort* -- port number obtained from a *serialBegin* opcode.

## See Also

`serialBegin`, `serialEnd`, `serialWrite_i`, `serialWrite`, `serialRead`, `serialPrint`.

## Credits

Author: Matt Ingalls  
2011

New in version 5.14

# serialPrint

serialPrint — Print data from a serial port.

## Description

Print to the screen any bytes (up to 32k) in the input buffer. Note that these bytes will be cleared from the buffer. use this opcode mainly for debugging messages. If you want to mix debugging and other communication messages over the same port, you will need to manually parse the data with the *serialRead* opcode.

## Syntax

```
serialPrint iPort
```

## Performance

*iPort* -- port number obtained from a *serialBegin* opcode.

## See Also

serialBegin, serialEnd, serialWrite\_i, serialWrite, serialRead, serialFlush.

## Credits

Author: Matt Ingalls  
2011

New in version 5.14

# serialRead

serialRead — Read data from a serial port.

## Description

Read data from a serial port for arduino.

## Syntax

```
kByte serialRead iPort
```

## Performance

*iPort* -- port number obtained from a *serialBegin* opcode.

*kByte* -- a byte of data to read.

## Examples

Here is an example of the serialRead opcode. It uses the file *serialRead.csd* [examples/serialRead.csd].

### Example 870. Example of the serialRead opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n ;;no output
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o serialRead.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 500 ; the default krate can be too fast for the arduino to handle
nchnls = 2
0dbfs = 1

instr 1

iPort serialBegin "/dev/cu.usbmodemfa131", 9600 ;connect to the arduino with baudrate = 9600

kGain init 16 ;read our knob value
kVal serialRead iPort
if (kVal != -1) then
    kGain = kVal/128
endif

aSig in ;get our audio input and get its rms
kRms rms aSig*kGain

kRms = kRms*kRms*255 ;scale the rms to a good value for the LED and send it out
```

```
serialWrite iPort, (kRms < 255 ? kRms : 255) ;must be in range: 0-255

endin
</CsInstruments>
<CsScore>
f 1 0 1024 10 1 1 1 1 1 1

i 1 0 200
e
</CsScore>
</CsoundSynthesizer>
```

This is the matching Arduino code :

```
void setup() {
  // enable serial communication
  Serial.begin(9600);

  // declare pin 9 to be an output:
  pinMode(9, OUTPUT);
}

void loop() {
  // only do something if we received something (this should be at csound's k-rate)
  if (Serial.available()) {
    // set the brightness of LED (connected to pin 9) to our input value
    int brightness = Serial.read();
    analogWrite(9, brightness);

    // while we are here, get our knob value and send it to csound
    int sensorValue = analogRead(A0);
    Serial.write(sensorValue/4); // scale to 1-byte range (0-255)
  }
}
.....
```

## See Also

[serialBegin](#), [serialEnd](#), [serialWrite\\_i](#), [serialWrite](#), [serialPrint](#), [serialFlush](#).

## Credits

Author: Matt Ingalls  
2011

New in version 5.14

# serialWrite\_i

`serialWrite_i` — Write data to a serial port.

## Description

Write data to a serial port for arduino.

## Syntax

```
serialWrite_i iPort, iByte  
serialWrite_i iPort, SBytes
```

## Initialization

*iPort* -- port number obtained from a *serialBegin* opcode.

*iByte* -- a byte of data to write.

## See Also

`serialBegin`, `serialEnd`, `serialWrite`, `serialRead`, `serialPrint`, `serialFlush`.

## Credits

Author: Matt Ingalls  
2011

New in version 5.14

# serialWrite

serialWrite — Write data to a serial port.

## Description

Write data to a serial port for arduino.

## Syntax

```
serialWrite iPort, iByte
serialWrite iPort, kByte
serialWrite iPort, SBytes
```

## Performance

*iPort* -- port number obtained from a *serialBegin* opcode.

*iByte* -- a byte of data to write.

## Examples

Here is an example of the serialWrite opcode. It uses the file *serialWrite.csd* [examples/serialWrite.csd].

### Example 871. Example of the serialWrite opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n ;;no output
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o serialWrite.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 500 ; the default krate can be too fast for the arduino to handle
nchnls = 2
0dbfs = 1

instr 1

iPort serialBegin "/dev/cu.usbmodemfa131", 9600 ;connect to the arduino with baudrate = 9600

kGain init 16 ;read our knob value
kVal serialRead iPort
if (kVal != -1) then
    kGain = kVal/128
endif
```

```
aSig in          ;get our audio input and get its rms
kRms rms aSig*kGain

kRms = kRms*kRms*255      ;scale the rms to a good value for the LED and send it out
serialWrite iPort, (kRms < 255 ? kRms : 255)    ;must be in range: 0-255

endin
</CsInstruments>
<CsScore>
f 1 0 1024 10 1 1 1 1 1 1

i 1 0 200
e
</CsScore>
</CsoundSynthesizer>
```

This is the matching Arduino code :

```
void setup() {
  // enable serial communication
  Serial.begin(9600);

  // declare pin 9 to be an output:
  pinMode(9, OUTPUT);
}

void loop() {
  // only do something if we received something (this should be at csound's k-rate)
  if (Serial.available()) {
    // set the brightness of LED (connected to pin 9) to our input value
    int brightness = Serial.read();
    analogWrite(9, brightness);

    // while we are here, get our knob value and send it to csound
    int sensorValue = analogRead(A0);
    Serial.write(sensorValue/4); // scale to 1-byte range (0-255)
  }
}
.....
```

## See Also

[serialBegin](#), [serialEnd](#), [serialWrite\\_i](#), [serialRead](#), [serialPrint](#), [serialFlush](#).

## Credits

Author: Matt Ingalls  
2011

New in version 5.14

# seqtime2

`seqtime2` — Generates a trigger signal according to the values stored in a table.

## Description

Generates a trigger signal according to the values stored in a table.

## Syntax

```
ktrig_out seqtime2 ktrig_in, ktime_unit, kstart, kloop, kinitndx, kfn_times
```

## Performance

*ktrig\_out* -- output trigger signal

*ktime\_unit* -- unit of measure of time, related to seconds.

*ktrig\_in* -- input trigger signal.

*kstart* -- start index of looped section

*kloop* -- end index of looped section

*kinitndx* -- initial index



### Note

Although *kinitndx* is listed as k-rate, it is in fact accessed only at init-time. So if you are using a k-rate argument, it must be assigned with *init*.

*kfn\_times* -- number of table containing a sequence of times

This opcode handles timed-sequences of groups of values stored into a table.

`seqtime2` generates a trigger signal (a sequence of impulses, see also `trigger` opcode), according to the values stored in the *kfn\_times* table. This table should contain a series of delta-times (i.e. times between to adjacent events). The time units stored into table are expressed in seconds, but can be rescaled by means of *ktime\_unit* argument. The table can be filled with `GEN02` or by means of an external text-file containing numbers, with `GEN23`.

It is possible to start the sequence from a value different than the first, by assigning to *initndx* an index different than zero (which corresponds to the first value of the table). Normally the sequence is looped, and the start and end of loop can be adjusted by modifying *kstart* and *kloop* arguments. User must be sure that values of these arguments (as well as *initndx*) correspond to valid table numbers, otherwise Csound will crash (because no range-checking is implemented).

It is possible to disable loop (one-shot mode) by assigning the same value both to *kstart* and *kloop* arguments. In this case, the last read element will be the one corresponding to the value of such arguments. Table can be read backward by assigning a negative *kloop* value. It is possible to trigger two events almost at the same time (actually separated by a k-cycle) by giving a zero value to the corresponding delta-time. First element contained in the table should be zero, if the user intends to send a trigger impulse, it should come immediately after the orchestra instrument containing `seqtime2` opcode.

*seqtime2* is similar to *seqtime*, the difference is that when *ktrig\_in* contains a non-zero value, current index is reset to *kinitndx* value. *kinitndx* can be varied at performance time.

## Examples

Here is an example of the *seqtime2* opcode. It uses the file *seqtime2.csd* [examples/seqtime2.csd].

### Example 872. Example of the *seqtime2* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o seqtime2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gitabMap2 ftgen 57,0,512,-2, 1,1/4,1/4,1/8,1/8,1/8,1/8,1/4,1/4,.5,1/4,1/4,1/16,1/16,1/16,1/16,1/16,1/16
gisine ftgen 1,0,512,10, 1

instr 1

ktrigin metro .333333333333
ktrig2 metro 1
  schedkwhen ktrig2, 0,0, 2, 0, .1 ; just to set the metronome!
kspeed init 1
; ktime_unit, kstart, kloop, initndx, kfn_times
ktrig seqtime2 ktrigin, kspeed, 0, 20, 2, gitabMap2
;ktrig seqtime kspeed, 0, 20, 0, gitabMap2 ; try with seqtime too...
  schedkwhen ktrig, 0, 0, 3, 0, ktrig ; the duration is got from seqtime2 output!
endin

instr 2

a1 line 1,p3,0
aout oscili 0.7*a1,500,gisine
  outs1 aout
endin

instr 3

a1 line 1,p3,0
aout oscili 0.7*a1,1000,gisine
  outs2 aout
endin

</CsInstruments>
<CsScore>
i1 0 20

;f0 3600
</CsScore>

```

`</CsoundSynthesizer>`

## See Also

*GEN02*, *GEN23*, *seqtime*, *trigseq*, *timedseq*

## Credits

Author: Gabriel Maldonado

# seqtime

seqtime — Generates a trigger signal according to the values stored in a table.

## Description

Generates a trigger signal according to the values stored in a table.

## Syntax

```
ktrig_out seqtime ktime_unit, kstart, kloop, kinitndx, kfn_times
```

## Performance

*ktrig\_out* -- output trigger signal

*ktime\_unit* -- unit of measure of time, related to seconds.

*kstart* -- start index of looped section

*kloop* -- end index of looped section

*kinitndx* -- initial index



### Note

Although *kinitndx* is listed as k-rate, it is in fact accessed only at init-time. So if you are using a k-rate argument, it must be assigned with *init*.

*kfn\_times* -- number of table containing a sequence of times

This opcode handles timed-sequences of groups of values stored into a table.

*seqtime* generates a trigger signal (a sequence of impulses, see also *trigger* opcode), according to the values stored in the *kfn\_times* table. This table should contain a series of delta-times (i.e. times between adjacent events). The time units stored into table are expressed in seconds, but can be rescaled by means of *ktime\_unit* argument. The table can be filled with *GEN02* or by means of an external text-file containing numbers, with *GEN23*.



### Note

Note that the *kloop* index marks the loop boundary and is NOT included in the looped elements. If you want to loop the first four elements, you would set *kstart* to 0 and *kloop* to 4.

It is possible to start the sequence from a value different than the first, by assigning to *kinitndx* an index different than zero (which corresponds to the first value of the table). Normally the sequence is looped, and the start and end of loop can be adjusted by modifying *kstart* and *kloop* arguments. User must be sure that values of these arguments (as well as *kinitndx*) correspond to valid table numbers, otherwise Csound will crash (because no range-checking is implemented).

It is possible to disable loop (one-shot mode) by assigning the same value both to *kstart* and *kloop* arguments. In this case, the last read element will be the one corresponding to the value of such arguments.

Table can be read backward by assigning a negative *kloop* value. It is possible to trigger two events almost at the same time (actually separated by a k-cycle) by giving a zero value to the corresponding delta-time. First element contained in the table should be zero, if the user intends to send a trigger impulse, it should come immediately after the orchestra instrument containing *seqtime* opcode.

## Examples

Here is an example of the *seqtime* opcode. It uses the file *seqtime.csd* [examples/seqtime.csd].

### Example 873. Example of the *seqtime* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>

<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o seqtime.wav -W ;; for file output any platform
</CsOptions>

<CsInstruments>

sr = 44100
ksmps = 64
nchnls = 1

; By Tim Mortimer and Andres Cabrera 2007

0dbfs = 1

gisine          ftgen      0, 0, 8192, 10,      1
;; table defining an integer pitch set
gipset          ftgen      0, 0, 4, -2, 8.00, 8.04, 8.07, 8.10
;;DELTA times for seqtime
gidelta         ftgen      0, 0, 4, -2, .5, 1, .25, 1.25

instr 1

kndx init 0
ktrigger init 0

ktime_unit init 1
kstart init p4
kloop init p5
kinitndx init 0
kfn_times init gidelta

ktrigger seqtime ktime_unit, kstart, kloop, kinitndx, kfn_times

printk2 ktrigger

if (ktrigger > 0) then
  kpitch table kndx, gipset
  event "i", 2, 0, 1, kpitch
  kndx = kndx + 1
  kndx = kndx % kloop
endif
endif
```

```
    endin

    instr 2
    icps = cspch (p4)
    a1   buzz   1, icps, 7, gisine
    aamp expseg 0.00003, .02, 1, p3-.02, 0.00003

    a1 = a1 * aamp * 0.5

    out a1
    endin

</CsInstruments>

<CsScore>
;   start   dur   kstart  kloop
i 1 0 7 0 4
i 1 8 10 0 3
i 1 19 10 4 4

</CsScore>

</CsoundSynthesizer>
```

## See Also

*GEN02*, *GEN23*, *trigseq seqtime2*

## Credits

Author: Gabriel Maldonado

November 2002. Added a note about the *kinitndx* parameter, thanks to Rasmus Ekman.

New in version 4.06

Example by: Tim Mortimer and Andres Cabrera 2007

# setctrl

setctrl — Configurable slider controls for realtime user input.

## Description

Configurable slider controls for realtime user input. Requires Winsound or TCL/TK. *setctrl* sets a slider to a specific value, or sets a minimum or maximum range.

## Syntax

```
setctrl inum, ival, itype
```

## Initialization

Note that this opcode is not available on Windows due to the implimentation of pipes on that system

*inum* -- number of the slider to set

*ival* -- value to be sent to the slider

*itype* -- type of value sent to the slider as follows:

- 1 -- set the current value. Initial value is 0.
- 2 -- set the minimum value. Default is 0.
- 3 -- set the maximum value. Default is 127.
- 4 -- set the label. (New in Csound version 4.09)

## Performance

Calling *setctrl* will create a new slider on the screen. There is no theoretical limit to the number of sliders. Windows and TCL/TK use only integers for slider values, so the values may need rescaling. GUIs usually pass values at a fairly slow rate, so it may be advisable to pass the output of control through *port*.

## Examples

Here is an example of the setctrl opcode. It uses the file *setctrl.csd* [examples/setctrl.csd].

### Example 874. Example of the setctrl opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
```

```
; -o setctrl.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Display the label "Volume" on Slider #1.
setctrl 1, "Volume", 4
; Set Slider #1's initial value to 20.
setctrl 1, 20, 1

; Capture and display the values for Slider #1.
k1 control 1
printk2 k1

; Play a simple oscillator.
; Use the values from Slider #1 for amplitude.
kamp = k1 * 128
a1 oscil kamp, 440, 1
out a1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for thirty seconds.
i 1 0 30
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
i1 38.00000
i1 40.00000
i1 43.00000
```

## See Also

*control*

## Credits

Author: John ffitch  
University of Bath, Codemist. Ltd.  
Bath, UK  
May 2000

Example written by Kevin Conder.

New in Csound version 4.06

# setksmps

setksmps — Sets the local ksmps value in an instrument or user-defined opcode block

## Description

Sets the local ksmps value in an instrument or user-defined opcode block.

## Syntax

```
setksmps iksmps
```

## Initialization

*iksmps* -- sets the local ksmps value.

If *iksmps* is set to zero, the *ksmps* of the caller instrument or opcode is used (this is the default behavior).



### Note

The local *ksmps* is implemented by splitting up a control period into smaller sub-kperiods and setting up the instrument local ksmps to the new value. This also requires converting the rate of k-rate input and output arguments (input variables receive the same value in all sub-kperiods, while outputs are written only in the last one). It also means that you cannot use a local *ksmps* that is higher than the global *ksmps*.



### Warning about local ksmps

When the local *ksmps* is not the same as the orchestra level *ksmps* value (as specified in the orchestra header), global a-rate operations must be carefully coded to access the data in a vector according to the local ksmps. The audio rate bus channel opcodes (chnget/chnset) can be used freely, however, as they will do the necessary conversion between ksmmp sizes.

Other opcodes that require some care include:

- any access to “ga” variables
- a-rate zak opcodes (*zar*, *zaw*, etc.)
- *tablera* and *tablewa* (these two opcodes may in fact work, but caution is needed)
- The *in* and *out* opcode family cannot be used in local-ksmps UDOs (these read from, and write to global a-rate buffers), but are safe in local-ksmps instruments

In general, the local *ksmps* should be used with care as it is an experimental feature. Though it works correctly in most cases.

The *setksmps* statement can be used to set the local *ksmps* value of the instrument or user-defined opcode block. It has one i-time parameter specifying the new *ksmps* value (which is left unchanged if zero is used). *setksmps* should be used before any other opcodes (but allowed after *xin* in UDOs), otherwise unpredictable results may occur.

## Performance

The syntax of a user-defined opcode block is as follows:

```
opcode name, outtypes, intypes
xinarg1 [, xinarg2] [, xinarg3] ... [xinargN] xin
[setksmps iksmps]
... the rest of the instrument's code.
xout xoutarg1 [, xoutarg2] [, xoutarg3] ... [xoutargN]
endop
```

The new opcode can then be used with the usual syntax:

```
[xinarg1] [, xinarg2] ... [xinargN] name [xoutarg1] [, xoutarg2] ... [xoutargN] [, iksmps]
```

## Examples

See the example for the *opcode* opcode.

## See Also

*endop*, *opcode*, *xin*, *xout*

## Credits

Author: Istvan Varga, 2002; based on code by Matt J. Ingalls; Modified by V Lazzarini for Csound 6

New in version 4.22

# setrow

setrow — Sets a given row of a 2-dimensional array from a vector.

## Description

Sets a given row of a 2-dimensional array. The output is an 2-d array with the contents of the requested row set to values of the input array (1-d; if 2-d the first row of the input array is used).

## Syntax

```
kout[] setrowkin[],krow
```

## Performance

*kout[]* -- output array containing the set row. It will be created if it does not exist.

*kin[]* -- input array.

*krow* -- row to be set.

## Examples

Here is an example of the setrow opcode. It uses the file *rfft.csd* [examples/rfft.csd].

### Example 875. Example of the setrow opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>

<CsOptions>
-d -o dac
</CsOptions>

<CsInstruments>
;ksmps needs to be an integer div of hopsize
ksmps = 64

instr 1

  ihopsize = 256 ; hopsize
  ifftsize = 1024 ; FFT size
  iolaps = ifftsize/ihopsize ; overlaps
  ibw = sr/ifftsize ; bin bandwidth
  kcnt init 0 ; counting vars
  krow init 0

  kOla[] init ifftsize ; overlap-add buffer
  kIn[] init ifftsize ; input buffer
  kSw[] init ifftsize
  kOut[][] init iolaps, ifftsize ; output buffers

  a1 diskin2 "fox.wav",1,0,1 ; audio input
```

```
ks expon 100, p3, 1000
asw vco2 k(a1), ks

/* every hopsize samples */
if kcnt == ihopsize then
  /* window and take FFT */
  kWin[] window kIn, krow*ihopsize
  kSpec[] rfft kWin
  kWin window kSw, krow*ihopsize
  kSpec2[] rfft kWin
  kProd[] cmplxprod kSpec, kSpec2

  /* IFFT + window */
  kRow[] rfft kProd
  kWin window kRow, krow*ihopsize
  /* place it on out buffer */
  kOut setrow kWin, krow

  /* zero the ola buffer */
  kOla = 0
  /* overlap-add */
  ki = 0
  until ki == iolaps do
    kRow getrow kOut, ki
    kOla = kOla + kRow
    ki += 1
  od

  /* update counters */
  krow = (krow+1)%iolaps
  kcnt = 0
endif

/* shift audio in/out of buffers */
kIn shiftin a1
kSw shiftin asw
a2 shiftout kOla
  out a2/iolaps

/* increment counter */
kcnt += ksmpls

endin

</CsInstruments>

<CsScore>
i1 0 10
</CsScore>

</CsoundSynthesizer>
```

## See Also

Vectorial opcodes array opcodes

## Credits

Author: Victor Lazzarini  
NUI Maynooth  
2014

New in version 6.04

# setscorepos

setscorepos — Sets the playback position of the current score performance to a given position.

## Description

Sets the playback position of the current score performance to a given position.

## Syntax

```
setscorepos ipos
```

## Initialization

*ipos* -- playback position in seconds.

## Examples

Here is an example of the setscorepos opcode. It uses the file *setscorepos.csd* [examples/setscorepos.csd].

### Example 876. Example of the setscorepos opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o setscorepos.wav -W   ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine ftgen 0, 0, 8192, 10, 1

instr 1

asig poscil 0.5, p4, giSine ;play something
outs asig, asig
endin

instr 11

setscorepos 8.5
endin

</CsInstruments>
<CsScore>
```

```
i1 0 2 220 ;this one will be played
i11 2.5 1 ;start setscorepos now
i1 3 2 330 ;skip this note
i1 6 2 440 ;and this one
i1 9 2 550 ;play this one

</CsScore>
</CsoundSynthesizer>
```

## See Also

*rewindscore*

## Credits

Author: Victor Lazzarini  
2008

New in Csound version 5.09

# sfilist

sfilist — Prints a list of all instruments of a previously loaded SoundFont2 (SF2) file.

## Description

Prints a list of all instruments of a previously loaded SoundFont2 (SF2) sample file. These opcodes allow management the sample-structure of SF2 files. In order to understand the usage of these opcodes, the user must have some knowledge of the SF2 format, so a brief description of this format can be found in the *SoundFont2 File Format Appendix*.

## Syntax

```
sfilist ifilhandle
```

## Initialization

*ifilhandle* -- unique number generated by *sload* opcode to be used as an identifier for a SF2 file. Several SF2 files can be loaded and activated at the same time.

## Performance

*sfilist* prints a list of all instruments of a previously loaded SF2 file to the console.

These opcodes only support the sample structure of SF2 files. The modulator structure of the SoundFont2 format is not supported in Csound. Any modulation or processing to the sample data is left to the Csound user, bypassing all restrictions forced by the SF2 standard.

## Examples

Here is an example of the sfilist opcode. It uses the file *sfilist.csd* [examples/sfilist.csd].

### Example 877. Example of the sfilist opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0   ;;realtime audio out, virtual midi in
;-iadc   ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o sfilist.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gisf sload "sf_GMbank.sf2"
```

```

sflist gisf      ;list all instruments

instr 1 ; play from score and midi keyboard

  mididefault 60, p3
  midinoteonkey p4, p5
  inum init p4
  ivel init p5
  ivel init ivel/127      ;make velocity dependent
  kamp linsegr 1, 1, 1, .1, 0
  kamp = kamp/3000      ;scale amplitude
  kfreq init 1      ;do not change freq from sf
  a1, a2 sfinstr3 ivel, inum, kamp*ivel, kfreq, 100, gisf ;choose Halo Pad
  outs a1, a2

endin

</CsInstruments>
<CsScore>
f0 60      ; stay active for 1 minute

i1 0 1 60 127
i1 + 1 62 <
i1 + 1 65 <
i1 + 1 69 10

e
</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like this:

```

Instrument list of "sf_GMbank.sf2"
  0) Piano 1
  1) Piano 2
  2) CP 70
  3) EP 1 layer 1
  4) EP 1 layer 2
  5) E.Piano 2
  6) Harpsichord
  .....
 100) Halo Pad
  .....

```

## See Also

*sfinstr*, *sfinstrm*, *sfload*, *sfpassign*, *sfplay*, *sfplaym*, *sfplist*, *sfpreset*

## Credits

Author: Gabriel Maldonado  
 Italy  
 May 2000

New in Csound Version 4.07

## **sfinstr3**

`sfinstr3` — Plays a SoundFont2 (SF2) sample instrument, generating a stereo sound with cubic interpolation.

### **Description**

Plays a SoundFont2 (SF2) sample instrument, generating a stereo sound with cubic interpolation. These opcodes allow management the sample-structure of SF2 files. In order to understand the usage of these opcodes, the user must have some knowledge of the SF2 format, so a brief description of this format can be found in the *SoundFont2 File Format Appendix*.

### **Syntax**

```
ar1, ar2 sfinstr3 ivel, inotenum, xamp, xfreq, instrnum, ifilhandle \  
[, iflag] [, ioffset]
```

### **Initialization**

*ivel* -- velocity value

*inotenum* -- MIDI note number value

*instrnum* -- number of an instrument of a SF2 file.

*ifilhandle* -- unique number generated by *sload* opcode to be used as an identifier for a SF2 file. Several SF2 files can be loaded and activated at the same time.

*iflag* (optional) -- flag regarding the behavior of *xfreq* and *inotenum*

*ioffset* (optional) -- start playing at offset, in samples.

### **Performance**

*xamp* -- amplitude correction factor

*xfreq* -- frequency value or frequency multiplier, depending by *iflag*. When *iflag* = 0, *xfreq* is a multiplier of a the default frequency, assigned by SF2 preset to the *inotenum* value. When *iflag* = 1, *xfreq* is the absolute frequency of the output sound, in Hz. Default is 0.

When *iflag* = 0, *inotenum* sets the frequency of the output according to the MIDI note number used, and *xfreq* is used as a multiplier. When *iflag* = 1, the frequency of the output, is set directly by *xfreq*. This allows the user to use any kind of micro-tuning based scales. However, this method is designed to work correctly only with presets tuned to the default equal temperament. Attempts to use this method with a preset already having non-standard tunings, or with drum-kit-based presets, could give unexpected results.

Adjustment of the amplitude can be done by varying the *xamp* argument, which acts as a multiplier.

The *ioffset* parameter allows the sound to start from a sample different than the first one. The user should make sure that its value is within the length of the specific sound. Otherwise, Csound will probably crash.

*sfinstr3* is a cubic-interpolation version of *sfinstr*. Difference of sound-quality is noticeable specially in bass-frequency-transposed samples. In high-freq-transposed samples the difference is less noticeable, and I suggest to use linear-interpolation versions, because they are faster.

These opcodes only support the sample structure of SF2 files. The modulator structure of the SoundFont2 format is not supported in Csound. Any modulation or processing to the sample data is left to the Csound user, bypassing all restrictions forced by the SF2 standard.

## Examples

Here is an example of the `sfinstr3` opcode. It uses the file `sfinstr3.csd` [examples/sfinstr3.csd].

### Example 878. Example of the `sfinstr3` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0 ;;realtime audio out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o sfinstr3.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gi24 ftgen 1, 0, 32, -2, 24, 2, 261.626, 60, 1, 1.0293022, 1.059463, 1.0905076, 1.1224619, 1.1553525,
1.2240532, 1.2599207, 1.2968391, 1.33483924, 1.3739531, 1.414213, 1.4556525, 1.4983063, 1.
1.6339145, 1.6817917, 1.73107, 1.7817962, 1.8340067, 1.8877471, 1.9430623, 2 ;table for m

giSF sfloat "sf_GMbank.sf2"
sfilist giSF

instr 1

mididefault 60, p3
midinoteonkey p4, p5
ikey = p4
ivel = p5
aenv linsegr 1, 1, 1, 1, 0 ;envelope
icps cpstuni ikey, 1 ;24 tones per octave
iamp = 0.0002 ;scale amplitude
iamp = iamp * ivel * 1/128 ;make velocity-dependent
aL, aR sfinstr3 ivel, ikey, iamp, icps, 180, giSF, 1 ;= Slap Bass 3
aL = aL * aenv
aR = aR * aenv
outs aL, aR

endin
</CsInstruments>
<CsScore>
f0 60 ;play for 60 seconds

i1 0 1 60 100 1 ;using ftable 1
i1 + 1 62 < .
i1 + 1 65 < .
i1 + 1 69 40 .

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*sfilist, sfinstr3m, sfinstrm, sfinstr, sfload, sfpassign, sfplay3, sfplay3m, sfplay, sfplaym, sfplist, sfpreset*

## Credits

Author: Gabriel Maldonado  
Italy  
May 2000

New in Csound Version 4.07

## sfinstr3m

`sfinstr3m` — Plays a SoundFont2 (SF2) sample instrument, generating a mono sound with cubic interpolation.

### Description

Plays a SoundFont2 (SF2) sample instrument, generating a mono sound with cubic interpolation. These opcodes allow management the sample-structure of SF2 files. In order to understand the usage of these opcodes, the user must have some knowledge of the SF2 format, so a brief description of this format can be found in the *SoundFont2 File Format Appendix*.

### Syntax

```
ares sfinstr3m ivel, inotenum, xamp, xfreq, instrnum, ifilhandle \  
    [, iflag] [, ioffset]
```

### Initialization

*ivel* -- velocity value

*inotenum* -- MIDI note number value

*instrnum* -- number of an instrument of a SF2 file.

*ifilhandle* -- unique number generated by *sload* opcode to be used as an identifier for a SF2 file. Several SF2 files can be loaded and activated at the same time.

*iflag* (optional) -- flag regarding the behavior of *xfreq* and *inotenum*

*ioffset* (optional) -- start playing at offset, in samples.

### Performance

*xamp* -- amplitude correction factor

*xfreq* -- frequency value or frequency multiplier, depending by *iflag*. When *iflag* = 0, *xfreq* is a multiplier of a the default frequency, assigned by SF2 preset to the *inotenum* value. When *iflag* = 1, *xfreq* is the absolute frequency of the output sound, in Hz. Default is 0.

When *iflag* = 0, *inotenum* sets the frequency of the output according to the MIDI note number used, and *xfreq* is used as a multiplier. When *iflag* = 1, the frequency of the output, is set directly by *xfreq*. This allows the user to use any kind of micro-tuning based scales. However, this method is designed to work correctly only with presets tuned to the default equal temperament. Attempts to use this method with a preset already having non-standard tunings, or with drum-kit-based presets, could give unexpected results.

Adjustment of the amplitude can be done by varying the *xamp* argument, which acts as a multiplier.

The *ioffset* parameter allows the sound to start from a sample different than the first one. The user should make sure that its value is within the length of the specific sound. Otherwise, Csound will probably crash.

*sfinstr3m* is a cubic-interpolation version of *sfinstrm*. Difference of sound-quality is noticeable specially in bass-frequency-transposed samples. In high-freq-transposed samples the difference is less noticeable, and I suggest to use linear-interpolation versions, because they are faster.

These opcodes only support the sample structure of SF2 files. The modulator structure of the SoundFont2 format is not supported in Csound. Any modulation or processing to the sample data is left to the Csound user, bypassing all restrictions forced by the SF2 standard.

## Examples

Here is an example of the *sfinstr3m* opcode. It uses the file *sfinstr3m.csd* [examples/sfinstr3m.csd].

### Example 879. Example of the *sfinstr3m* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0 ;;realtime audio out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o sfinstr3m.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gisf sload "07AcousticGuitar.sf2"
sfilist isf

instr 1 ; play from score and midi keyboard

mididefault 60, p3
midinoteonkey p4, p5
inum init p4
ivel init p5
ivel init ivel/127 ;make velocity dependent
kamp linsegr 1, 1, 1, .1, 0
kamp = kamp/7000 ;scale amplitude
kfreq init 1 ;do not change freq from sf
aout sfinstr3m ivel, inum, kamp*ivel, kfreq, 0, gisf
outs aout, aout

endin

</CsInstruments>
<CsScore>
f0 60 ; stay active for 1 minute

i1 0 1 60 127
i1 + 1 62 <
i1 + 1 65 <
i1 + 1 69 10

e
</CsScore>
```

`</CsoundSynthesizer>`

## See Also

*sfilist, sfinstr3, sfinstr, sfinstrm, sfload, sfpassign, sfplay3, sfplay3m, sfplay, sfplaym, sfplist, sfpreset*

## Credits

Author: Gabriel Maldonado  
Italy  
May 2000

New in Csound Version 4.07

# sfinstr

sfinstr — Plays a SoundFont2 (SF2) sample instrument, generating a stereo sound.

## Description

Plays a SoundFont2 (SF2) sample instrument, generating a stereo sound. These opcodes allow management the sample-structure of SF2 files. In order to understand the usage of these opcodes, the user must have some knowledge of the SF2 format, so a brief description of this format can be found in the *SoundFont2 File Format Appendix*.

## Syntax

```
ar1, ar2 sfinstr ivel, inotenum, xamp, xfreq, instrnum, ifilhandle \  
[, iflag] [, ioffset]
```

## Initialization

*ivel* -- velocity value

*inotenum* -- MIDI note number value

*instrnum* -- number of an instrument of a SF2 file.

*ifilhandle* -- unique number generated by *sload* opcode to be used as an identifier for a SF2 file. Several SF2 files can be loaded and activated at the same time.

*iflag* (optional) -- flag regarding the behavior of *xfreq* and *inotenum*

*ioffset* (optional) -- start playing at offset, in samples.

## Performance

*xamp* -- amplitude correction factor

*xfreq* -- frequency value or frequency multiplier, depending by *iflag*. When *iflag* = 0, *xfreq* is a multiplier of a the default frequency, assigned by SF2 preset to the *inotenum* value. When *iflag* = 1, *xfreq* is the absolute frequency of the output sound, in Hz. Default is 0.

When *iflag* = 0, *inotenum* sets the frequency of the output according to the MIDI note number used, and *xfreq* is used as a multiplier. When *iflag* = 1, the frequency of the output, is set directly by *xfreq*. This allows the user to use any kind of micro-tuning based scales. However, this method is designed to work correctly only with presets tuned to the default equal temperament. Attempts to use this method with a preset already having non-standard tunings, or with drum-kit-based presets, could give unexpected results.

Adjustment of the amplitude can be done by varying the *xamp* argument, which acts as a multiplier.

The *ioffset* parameter allows the sound to start from a sample different than the first one. The user should make sure that its value is within the length of the specific sound. Otherwise, Csound will probably crash.

*sfinstr* plays an SF2 instrument instead of a preset (an SF2 instrument is the base of a preset layer). *instrnum* specifies the instrument number, and the user must be sure that the specified number belongs to an existing

instrument of a determinate soundfont bank. Notice that both *xamp* and *xfreq* can operate at k-rate as well as a-rate, but both arguments must work at the same rate.

These opcodes only support the sample structure of SF2 files. The modulator structure of the SoundFont2 format is not supported in Csound. Any modulation or processing to the sample data is left to the Csound user, bypassing all restrictions forced by the SF2 standard.

## Examples

Here is an example of the *sfinstr* opcode. It uses the file *sfinstr.csd* [examples/sfinstr.csd].

### Example 880. Example of the *sfinstr* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0 ;;realtime audio out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o sfinstr.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gisf sload "sf_GMbank.sf2"
sfilist isf

instr 1 ; play from score and midi keyboard

mididefault 60, p3
midinoteonkey p4, p5
inum init p4
ivel init p5
ivel init ivel/127 ;make velocity dependent
kamp linsegr 1, 1, 1, .1, 0
kamp = kamp/5000 ;scale amplitude
kfreq init 1 ;do not change freq from sf
a1,a2 sfinstr ivel, inum, kamp*ivel, kfreq, 194, gisf ;= Strings 2 tighter
outs a1, a2

endin

</CsInstruments>
<CsScore>
f0 60 ; stay active for 1 minute

i1 0 1 60 127
i1 + 1 62 <
i1 + 1 65 <
i1 + 1 69 10

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*sfilist, sfinstrm, sfload, sfpassign, sfplay, sfplaym, sfplist, sfpreset*

## Credits

Author: Gabriel Maldonado  
Italy  
May 2000

New in Csound Version 4.07

# sfinstrm

`sfinstrm` — Plays a SoundFont2 (SF2) sample instrument, generating a mono sound.

## Description

Plays a SoundFont2 (SF2) sample instrument, generating a mono sound. These opcodes allow management the sample-structure of SF2 files. In order to understand the usage of these opcodes, the user must have some knowledge of the SF2 format, so a brief description of this format can be found in the *SoundFont2 File Format Appendix*.

## Syntax

```
ares sfinstrm ivel, inotenum, xamp, xfreq, instrnum, ifilhandle \  
    [, iflag] [, ioffset]
```

## Initialization

*ivel* -- velocity value

*inotenum* -- MIDI note number value

*instrnum* -- number of an instrument of a SF2 file.

*ifilhandle* -- unique number generated by *sload* opcode to be used as an identifier for a SF2 file. Several SF2 files can be loaded and activated at the same time.

*iflag* (optional) -- flag regarding the behavior of *xfreq* and *inotenum*

*ioffset* (optional) -- start playing at offset, in samples.

## Performance

*xamp* -- amplitude correction factor

*xfreq* -- frequency value or frequency multiplier, depending by *iflag*. When *iflag* = 0, *xfreq* is a multiplier of a the default frequency, assigned by SF2 preset to the *inotenum* value. When *iflag* = 1, *xfreq* is the absolute frequency of the output sound, in Hz. Default is 0.

When *iflag* = 0, *inotenum* sets the frequency of the output according to the MIDI note number used, and *xfreq* is used as a multiplier. When *iflag* = 1, the frequency of the output, is set directly by *xfreq*. This allows the user to use any kind of micro-tuning based scales. However, this method is designed to work correctly only with presets tuned to the default equal temperament. Attempts to use this method with a preset already having non-standard tunings, or with drum-kit-based presets, could give unexpected results.

Adjustment of the amplitude can be done by varying the *xamp* argument, which acts as a multiplier.

The *ioffset* parameter allows the sound to start from a sample different than the first one. The user should make sure that its value is within the length of the specific sound. Otherwise, Csound will probably crash.

*sfinstrm* is a mono version of *sfinstr*. This is the fastest opcode of the SF2 family.

These opcodes only support the sample structure of SF2 files. The modulator structure of the SoundFont2 format is not supported in Csound. Any modulation or processing to the sample data is left to the Csound user, bypassing all restrictions forced by the SF2 standard.

## See Also

*sfilist, sfinstr, sfload, sfpassign, sfplay, sfplaym, sfplist, sfpreset*

## Credits

Author: Gabriel Maldonado  
Italy  
May 2000

New in Csound Version 4.07

# sfload

`sfload` — Loads an entire SoundFont2 (SF2) sample file into memory.

## Description

Loads an entire SoundFont2 (SF2) sample file into memory. These opcodes allow management the sample-structure of SF2 files. In order to understand the usage of these opcodes, the user must have some knowledge of the SF2 format, so a brief description of this format can be found in the *SoundFont2 File Format Appendix*.

`sfload` should be placed in the header section of a Csound orchestra.

## Syntax

```
ir sfload "filename"
```

## Initialization

`ir` -- output to be used by other SF2 opcodes. For `sfload`, `ir` is `ifilhandle`.

`"filename"` -- name of the SF2 file, with its complete path. It must be a string typed within double-quotes with `/` to separate directories (this applies to DOS and Windows as well, where using a backslash will generate an error), or an integer that has been the subject of a `strset` operation

## Performance

`sfload` loads an entire SF2 file into memory. It returns a file handle to be used by other opcodes. Several instances of `sfload` can be placed in the header section of an orchestra, allowing use of more than one SF2 file in a single orchestra.

These opcodes only support the sample structure of SF2 files. The modulator structure of the SoundFont2 format is not supported in Csound. Any modulation or processing to the sample data is left to the Csound user, bypassing all restrictions forced by the SF2 standard.

It should be noted that before version 5.12 a maximum of 10 sound fonts could be loaded, a restriction since relaxed.

## Examples

Here is an example of the `sfload` opcode. It uses the file `sfload.csd` [examples/sfload.csd].

### Example 881. Example of the `sfload` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0   ;;realtime audio out, virtual midi in
;-iadc   ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o sfload.wav -W   ;; for file output any platform
```

```

</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

;load two soundfonts
isf sfload "07AcousticGuitar.sf2"
ir sfload "01hpschd.sf2"
sfplist isf
sfplist ir
sfpassign 0, isf
sfpassign 1, ir

instr 1 ; play guitar from score and midi keyboard - preset index = 0

mididefault 60, p3
midinoteonkey p4, p5
inum init p4
ivel init p5
ivel init ivel/127 ;make velocity dependent
kamp linsegr 1, 1, 1, .1, 0
kamp = kamp/3000 ;scale amplitude
kfreq init 1 ;do not change freq from sf
a1,a2 sfplay3 ivel, inum, kamp*ivel, kfreq, 0 ;preset index = 0
outs a1, a2

endin

instr 2 ; play harpsichord from score and midi keyboard - preset index = 1

mididefault 60, p3
midinoteonkey p4, p5
inum init p4
ivel init p5
ivel init ivel/127 ;make velocity dependent
kamp linsegr 1, 1, 1, .1, 0
kamp = kamp/1000 ;scale amplitude
kfreq init 1 ;do not change freq from sf
a1,a2 sfplay3 ivel, inum, kamp*ivel, kfreq, 1 ;preset index = 1
outs a1, a2

endin

</CsInstruments>
<CsScore>
f0 60 ; stay active for 1 minute

i1 0 1 60 100
i1 + 1 62 <
i1 + 1 65 <
i1 + 1 69 10

i2 5 1 60 100
i2 + 1 62 <
i2 7 1 65 <
i2 7 1 69 10

e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*sfilist, sfinstr, sfinstrm, sfpassign, sfplay, sfplaym, sfplist, sfpreset*

## Credits

Author: Gabriel Maldonado  
Italy  
May 2000

New in Csound Version 4.07

# sflooper

`sflooper` — Plays a SoundFont2 (SF2) sample preset, generating a stereo sound, with user-defined time-varying crossfade looping.

## Description

Plays a SoundFont2 (SF2) sample preset, generating a stereo sound, similarly to `sfplay`. Unlike that opcode, though, it ignores the looping points set in the SF2 file and substitutes them for a user-defined crossfade loop. It is a cross between `sfplay` and `flooper2`.

## Syntax

```
ar1, ar2 sflooper ivel, inotenum, kamp, kpitch, ipreindex, kloopstart, kloopend, kcrossfade \  
[, istart, imode, ifenv, iskip]
```

## Initialization

*ivel* -- velocity value

*inotenum* -- MIDI note number value

*ipreindex* -- preset index

*istart* -- playback start pos in seconds

*imode* -- loop modes: 0 forward, 1 backward, 2 back-and-forth [def: 0]

*ifenv* -- if non-zero, crossfade envelope shape table number. The default, 0, sets the crossfade to linear.

*iskip* -- if 1, the opcode initialisation is skipped, for tied notes, performance continues from the position in the loop where the previous note stopped. The default, 0, does not skip initialisation

## Performance

*kamp* -- amplitude scaling

*kpitch* -- pitch control (transposition ratio); negative values are not allowed.

*kloopstart* -- loop start point (secs). Note that although k-rate, loop parameters such as this are only updated once per loop cycle. If loop start is set beyond the end of the sample, no looping will result.

*kloopend* -- loop end point (secs), updated once per loop cycle.

*kcrossfade* -- crossfade length (secs), updated once per loop cycle and limited to loop length.

*sflooper* plays a preset, generating a stereo sound.

These opcodes only support the sample structure of SF2 files. The modulator structure of the SoundFont2 format is not supported in Csound. Any modulation or processing to the sample data is left to the Csound user, bypassing all restrictions forced by the SF2 standard.

Note: The looping points are set on the root key of every sample that is part of the preset of the soundfont. For instance, a soundfont can have a single sample for the whole keyboard. In that case, sflooper will work like flooper and flooper2, because as the sample is transposed, played back at different rates, the loop will get short or longer. If however the soundfont has a sample for each key, than there will be no transposition and the loop will stay the same length (unless you change kpitch).

## Examples

Here is an example of the sflooper opcode. It uses the file *sflooper.csd* [examples/sflooper.csd].

### Example 882. Example of the sflooper opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac -+rtmidi=virtual -M0      ;;realtime audio in, midi in
; For Non-realtime ouput leave only the line below:
; -o sflooper.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

isf  sfload "07AcousticGuitar.sf2"
     sfpassign 0, isf

instr 1 ; play from score and midi keyboard

     mididefault 60, p3
     midinoteonkey p4, p5
inum  init p4
ivel  init p5
print ivel

ivel  init  ivel/127 ;velocity dependent
kamp  linsegr 1,1,1,.1,0 ;envelope
kamp  = kamp * .0002 ;scale amplitude (= kamp/5000)
kfreq  init 1 ;do not change freq from sf
;"07AcousticGuitar.sf2" contains 2 samples, on notes E1 and C#4
;start loop from beginning, loop .2 seconds - on the root key of these samples
aL,aR  sflooper ivel, inum, kamp*ivel, kfreq, 0, 0, .2, .05
     outs aL, aR

endin
</CsInstruments>
<CsScore>
f0 60 ; stay active for 1 minute

i1 0 1 60 100
i1 + 1 62 <
i1 + 1 65 <
i1 + 1 69 10
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*sfilist, sfinstr, sfinstrm, sfload, sfpassign, sfplaym, sfplist, sfpreset*

## Credits

Author: Victor Lazzarini  
August 2007

New in Csound Version 5.07

# sfpassign

`sfpassign` — Assigns all presets of a SoundFont2 (SF2) sample file to a sequence of progressive index numbers.

## Description

Assigns all presets of a previously loaded SoundFont2 (SF2) sample file to a sequence of progressive index numbers. These opcodes allow management the sample-structure of SF2 files. In order to understand the usage of these opcodes, the user must have some knowledge of the SF2 format, so a brief description of this format can be found in the *SoundFont2 File Format Appendix*.

`sfpassign` should be placed in the header section of a Csound orchestra.

## Syntax

```
sfpassign istartindex, ifilhandle[, imsgs]
```

## Initialization

`istartindex` -- starting index preset by the user in bulk preset assignments.

`ifilhandle` -- unique number generated by `sfload` opcode to be used as an identifier for a SF2 file. Several SF2 files can be loaded and activated at the same time.

`imsgs` -- if non-zero messages are suppressed.

## Performance

`sfpassign` assigns all presets of a previously loaded SF2 file to a sequence of progressive index numbers, to be used later with the opcodes `sfplay` and `sfplaym`. `istartindex` specifies the starting index number. Any number of `sfpassign` instances can be placed in the header section of an orchestra, each one assigning presets belonging to different SF2 files. The user must take care that preset index numbers of different SF2 files do not overlap.

These opcodes only support the sample structure of SF2 files. The modulator structure of the SoundFont2 format is not supported in Csound. Any modulation or processing to the sample data is left to the Csound user, bypassing all restrictions forced by the SF2 standard.

## Examples

Here is an example of the `sfpassign` opcode. It uses the file `sfpassign.csd` [examples/sfpassign.csd].

### Example 883. Example of the `sfpassign` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0    ;;realtime audio out, virtual midi in
;-iadc    ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
```

```

; -o sfpassign.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

;load two soundfonts
gisf sflload "07AcousticGuitar.sf2"
gir sflload "01hpschd.sf2"
sfplist gisf
sfplist gir
sfpassign 0, gisf
sfpassign 1, gir

instr 1 ; play guitar from score and midi keyboard - preset index = 0

mididefault 60, p3
midinoteonkey p4, p5
inum init p4
ivel init p5
ivel init ivel/127 ;make velocity dependent
kamp linsegr 1, 1, 1, .1, 0
kamp = kamp/5000 ;scale amplitude
kfreq init 1 ;do not change freq from sf
a1,a2 sfplay3 ivel, inum, kamp*ivel, kfreq, 0 ;preset index = 0
outs a1, a2

endin

instr 2 ; play harpsichord from score and midi keyboard - preset index = 1

mididefault 60, p3
midinoteonkey p4, p5
inum init p4
ivel init p5
ivel init ivel/127 ;make velocity dependent
kamp linsegr 1, 1, 1, .1, 0
kamp = kamp/1000 ;scale amplitude
kfreq init 1 ;do not change freq from sf
a1,a2 sfplay3 ivel, inum, kamp*ivel, kfreq, 1 ;preset index = 1
outs a1, a2

endin

</CsInstruments>
<CsScore>
f0 60 ; stay active for 1 minute

i1 0 1 60 100
i1 + 1 62 <
i1 + 1 65 <
i1 + 1 69 10

i2 5 1 60 100
i2 + 1 62 <
i2 7 1 65 <
i2 7 1 69 10

e
</CsScore>
</CsSoundSynthesizer>

```

## See Also

*sfilist, sfinstr, sfinstrm, sfload, sfplay, sfplaym, sfplist, sfpreset*

## Credits

Author: Gabriel Maldonado  
Italy  
May 2000

New in Csound Version 4.07

# sfplay3

`sfplay3` — Plays a SoundFont2 (SF2) sample preset, generating a stereo sound with cubic interpolation.

## Description

Plays a SoundFont2 (SF2) sample preset, generating a stereo sound with cubic interpolation. These opcodes allow management the sample-structure of SF2 files. In order to understand the usage of these opcodes, the user must have some knowledge of the SF2 format, so a brief description of this format can be found in the *SoundFont2 File Format Appendix*.

## Syntax

```
ar1, ar2 sfplay3 ivel, inotenum, xamp, xfreq, ipreindex [, iflag] [, ioffset] [, ienv]
```

## Initialization

*ivel* -- velocity value

*inotenum* -- MIDI note number value

*ipreindex* -- preset index

*iflag* (optional) -- flag regarding the behavior of *xfreq* and *inotenum*

*ioffset* (optional) -- start playing at offset, in samples.

*ienv* (optional) -- enables and determines amplitude envelope. 0 = no envelope, 1 = linear attack and decay, 2 = linear attack, exponential decay (see below). Default = 0.

## Performance

*xamp* -- amplitude correction factor

*xfreq* -- frequency value or frequency multiplier, depending by *iflag*. When *iflag* = 0, *xfreq* is a multiplier of a the default frequency, assigned by SF2 preset to the *inotenum* value. When *iflag* = 1, *xfreq* is the absolute frequency of the output sound, in Hz. Default is 0.

When *iflag* = 0, *inotenum* sets the frequency of the output according to the MIDI note number used, and *xfreq* is used as a multiplier. When *iflag* = 1, the frequency of the output, is set directly by *xfreq*. This allows the user to use any kind of micro-tuning based scales. However, this method is designed to work correctly only with presets tuned to the default equal temperament. Attempts to use this method with a preset already having non-standard tunings, or with drum-kit-based presets, could give unexpected results.

Adjustment of the amplitude can be done by varying the *xamp* argument, which acts as a multiplier.

Notice that both *xamp* and *xfreq* can use k-rate as well as a-rate signals. Both arguments must use variables of the same rate, or *sfplay3* will not work correctly. *ipreindex* must contain the number of a previously assigned preset, or Csound will crash.

The *ioffset* parameter allows the sound to start from a sample different than the first one. The user should make sure that its value is within the length of the specific sound. Otherwise, Csound will probably crash.

The *ienv* parameter enables and determines the type of amplitude envelope used. The default value is 0, or no envelope. If *ienv* is set to 1, the attack and decay portions are linear. If set to 2, the attack is linear and the decay is exponential. The release portion of the envelope has not yet been implemented.

*sfplay3* plays a preset, generating a stereo sound with cubic interpolation. *ivel* does not directly affect the amplitude of the output, but informs *sfplay3* about which sample should be chosen in multi-sample, velocity-split presets.

*sfplay3* is a cubic-interpolation version of *sfplay*. Difference of sound-quality is noticeable specially in bass-frequency-transposed samples. In high-freq-transposed samples the difference is less noticeable, and I suggest to use linear-interpolation versions, because they are faster.

These opcodes only support the sample structure of SF2 files. The modulator structure of the SoundFont2 format is not supported in Csound. Any modulation or processing to the sample data is left to the Csound user, bypassing all restrictions forced by the SF2 standard.

## Examples

Here is an example of the *sfplay3* opcode. It uses the file *sfplay3.csd* [examples/sfplay3.csd].

### Example 884. Example of the *sfplay3* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0 ;;realtime audio out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o sfplay3.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gitwelve ftgen 1, 0, 16, -2, 12, 2, 440, 69, 1, 16/15, 9/8, 6/5, 5/4, 4/3, 7/5, 3/2, 8/5, 5/3, 9/5, 15/8
givife ftgen 2, 0, 16, -2, 5, 2, 261.659, 60, 1, 1.1486, 1.3195, 1.5157, 1.7411, 2.00

giSF sfload "01hpschd.sf2"
sfplist giSF
gipre sfpreset 0, 0, giSF, 0

instr 1

mididefault 60, p3
midinoteonkey p4, p5
ikey = p4
ivel = p5
aenv linsegr 1, 1, 1, 1, 0 ;envelope
icps cpstuni ikey, gitwelve ;12 tones per octave
iamp = 0.0004 ;scale amplitude
iamp = iamp * ivel * 1/128 ;make velocity-dependent
aL, aR sfplay3 ivel, ikey, iamp, icps, gipre, 1
aL = aL * aenv
aR = aR * aenv
outs aL, aR
```

```
endin

instr 2

  mididefault 60, p3
  midinoteonkey p4, p5
  ikey = p4
  ivel = p5
  aenv linsegr 1, 1, 1, 1, 0 ;envelope
  icps cpstuni ikey, givife ;5 tones per octave
  iamp = 0.0004 ;scale amplitude
  iamp = iamp * ivel * 1/128 ;make velocity-dependent
  aL, aR sfplay3 ivel, ikey, iamp, icps, gipre, 1
  aL = aL * aenv
  aR = aR * aenv
  outs aL, aR

endin
</CsInstruments>
<CsScore>
f0 60 ;play for 60 seconds
;instr.1 using ftable 1
i1 0 1 60 100
i1 + 1 62 <
i1 + 1 65 <
i1 + 1 69 40

;instr.2 using ftable 2
i2 5 1 60 100
i2 + 1 62 <
i2 + 1 65 <
i2 + 1 69 40
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*sfilist, sfinstr3, sfinstr3m, sfinstr, sfinstrm, sfload, sfpassign, sfplay3m, sfplaym, sfplay, sfplist, sfpreset*

## Credits

Author: Gabriel Maldonado  
Italy  
May 2000

New in Csound Version 4.07

New optional parameter *ienv* in version 5.09

# sfplay3m

`sfplay3m` — Plays a SoundFont2 (SF2) sample preset, generating a mono sound with cubic interpolation.

## Description

Plays a SoundFont2 (SF2) sample preset, generating a mono sound with cubic interpolation. These opcodes allow management the sample-structure of SF2 files. In order to understand the usage of these opcodes, the user must have some knowledge of the SF2 format, so a brief description of this format can be found in the *SoundFont2 File Format Appendix*.

## Syntax

```
ares sfplay3m ivel, inotenum, xamp, xfreq, ipreindex [, iflag] [, ioffset] [, ienv]
```

## Initialization

*ivel* -- velocity value

*inotenum* -- MIDI note number value

*ipreindex* -- preset index

*iflag* (optional) -- flag regarding the behavior of *xfreq* and *inotenum*

*ioffset* (optional) -- start playing at offset, in samples.

*ienv* (optional) -- enables and determines amplitude envelope. 0 = no envelope, 1 = linear attack and decay, 2 = linear attack, exponential decay (see below). Default = 0.

## Performance

*xamp* -- amplitude correction factor

*xfreq* -- frequency value or frequency multiplier, depending by *iflag*. When *iflag* = 0, *xfreq* is a multiplier of a the default frequency, assigned by SF2 preset to the *inotenum* value. When *iflag* = 1, *xfreq* is the absolute frequency of the output sound, in Hz. Default is 0.

When *iflag* = 0, *inotenum* sets the frequency of the output according to the MIDI note number used, and *xfreq* is used as a multiplier. When *iflag* = 1, the frequency of the output, is set directly by *xfreq*. This allows the user to use any kind of micro-tuning based scales. However, this method is designed to work correctly only with presets tuned to the default equal temperament. Attempts to use this method with a preset already having non-standard tunings, or with drum-kit-based presets, could give unexpected results.

Adjustment of the amplitude can be done by varying the *xamp* argument, which acts as a multiplier.

Notice that both *xamp* and *xfreq* can use k-rate as well as a-rate signals. Both arguments must use variables of the same rate, or *sfplay3m* will not work correctly. *ipreindex* must contain the number of a previously assigned preset, or Csound will crash.

The *ioffset* parameter allows the sound to start from a sample different than the first one. The user should make sure that its value is within the length of the specific sound. Otherwise, Csound will probably crash.

The *ienv* parameter enables and determines the type of amplitude envelope used. The default value is 0, or no envelope. If *ienv* is set to 1, the attack and decay portions are linear. If set to 2, the attack is linear and the decay is exponential. The release portion of the envelope has not yet been implemented.

*sfplay3m* is a mono version of *sfplay3*. It should be used with mono preset, or with the stereo presets in which stereo output is not required. It is faster than *sfplay3*.

*sfplay3m* is also a cubic-interpolation version of *sfplaym*. Difference of sound-quality is noticeable specially in bass-frequency-transposed samples. In high-freq-transposed samples the difference is less noticeable, and I suggest to use linear-interpolation versions, because they are faster.

These opcodes only support the sample structure of SF2 files. The modulator structure of the SoundFont2 format is not supported in Csound. Any modulation or processing to the sample data is left to the Csound user, bypassing all restrictions forced by the SF2 standard.

## Examples

Here is an example of the *sfplay3m* opcode. It uses the file *sfplay3m.csd* [examples/sfplay3m.csd].

### Example 885. Example of the *sfplay3m* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0 ;;realtime audio out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o sfplay3m.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gisf sfloat "07AcousticGuitar.sf2"
sfplist gisf
sfpassign 10, gisf

instr 1 ; play from score and midi keyboard

mididefault 60, p3
midinoteonkey p4, p5
inum init p4
ivel init p5
ivel init ivel/127 ;make velocity dependent
kamp linsegr 1, 1, 1, .1, 0
kamp = kamp/7000 ;scale amplitude
kfreq init 1 ;do not change freq from sf
out sfplay3m ivel, inum, kamp*ivel, kfreq, 10 ;preset index = 10
outs out, out

endin

</CsInstruments>
<CsScore>
```

```
f0 60 ; stay active for 1 minute

i1 0 1 60 127
i1 + 1 62 <
i1 + 1 65 <
i1 + 1 69 10

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*sfilist, sfinstr3, sfinstr3m, sfinstr, sfinstrm, sfload, sfpassign, sfplay3, sfplaym, sfplay, sfplist, sfpreset*

## Credits

Author: Gabriel Maldonado  
Italy  
May 2000

New in Csound Version 4.07

New optional parameter *ienv* in version 5.09

# sfplay

sfplay — Plays a SoundFont2 (SF2) sample preset, generating a stereo sound.

## Description

Plays a SoundFont2 (SF2) sample preset, generating a stereo sound. These opcodes allow management the sample-structure of SF2 files. In order to understand the usage of these opcodes, the user must have some knowledge of the SF2 format, so a brief description of this format can be found in the *SoundFont2 File Format Appendix*.

## Syntax

```
ar1, ar2 sfplay ivel, inotenum, xamp, xfreq, ipreindex [, iflag] [, ioffset] [, ienv]
```

## Initialization

*ivel* -- velocity value

*inotenum* -- MIDI note number value

*ipreindex* -- preset index

*iflag* (optional) -- flag regarding the behavior of *xfreq* and *inotenum*

*ioffset* (optional) -- start playing at offset, in samples.

*ienv* (optional) -- enables and determines amplitude envelope. 0 = no envelope, 1 = linear attack and decay, 2 = linear attack, exponential decay (see below). Default = 0.

## Performance

*xamp* -- amplitude correction factor

*xfreq* -- frequency value or frequency multiplier, depending by *iflag*. When *iflag* = 0, *xfreq* is a multiplier of a the default frequency, assigned by SF2 preset to the *inotenum* value. When *iflag* = 1, *xfreq* is the absolute frequency of the output sound, in Hz. Default is 0.

When *iflag* = 0, *inotenum* sets the frequency of the output according to the MIDI note number used, and *xfreq* is used as a multiplier. When *iflag* = 1, the frequency of the output, is set directly by *xfreq*. This allows the user to use any kind of micro-tuning based scales. However, this method is designed to work correctly only with presets tuned to the default equal temperament. Attempts to use this method with a preset already having non-standard tunings, or with drum-kit-based presets, could give unexpected results.

Adjustment of the amplitude can be done by varying the *xamp* argument, which acts as a multiplier.

Notice that both *xamp* and *xfreq* can use k-rate as well as a-rate signals. Both arguments must use variables of the same rate, or *sfplay* will not work correctly. *ipreindex* must contain the number of a previously assigned preset, or Csound will crash.

The *ioffset* parameter allows the sound to start from a sample different than the first one. The user should make sure that its value is within the length of the specific sound. Otherwise, Csound will probably crash.

The *ienv* parameter enables and determines the type of amplitude envelope used. The default value is 0, or no envelope. If *ienv* is set to 1, the attack and decay portions are linear. If set to 2, the attack is linear and the decay is exponential. The release portion of the envelope has not yet been implemented.

*sfplay* plays a preset, generating a stereo sound. *ivel* does not directly affect the amplitude of the output, but informs *sfplay* about which sample should be chosen in multi-sample, velocity-split presets.

These opcodes only support the sample structure of SF2 files. The modulator structure of the SoundFont2 format is not supported in Csound. Any modulation or processing to the sample data is left to the Csound user, bypassing all restrictions forced by the SF2 standard.

## See Also

*sfilist*, *sfinstr*, *sfinstrm*, *sfload*, *sfpassign*, *sfplay3*, *sfplaym*, *sfplay3m*, *sfplist*, *sfpreset*

## Credits

Author: Gabriel Maldonado  
Italy  
May 2000

New in Csound Version 4.07

New optional parameter *ienv* in version 5.09

# sfplaym

sfplaym — Plays a SoundFont2 (SF2) sample preset, generating a mono sound.

## Description

Plays a SoundFont2 (SF2) sample preset, generating a mono sound. These opcodes allow management the sample-structure of SF2 files. In order to understand the usage of these opcodes, the user must have some knowledge of the SF2 format, so a brief description of this format can be found in the *SoundFont2 File Format Appendix*.

## Syntax

```
ares sfplaym ivel, inotenum, xamp, xfreq, ipreindex [, iflag] [, ioffset] [, ienv]
```

## Initialization

*ivel* -- velocity value

*inotenum* -- MIDI note number value

*ipreindex* -- preset index

*iflag* (optional) -- flag regarding the behavior of *xfreq* and *inotenum*

*ioffset* (optional) -- start playing at offset, in samples.

*ienv* (optional) -- enables and determines amplitude envelope. 0 = no envelope, 1 = linear attack and decay, 2 = linear attack, exponential decay (see below). Default = 0.

## Performance

*xamp* -- amplitude correction factor

*xfreq* -- frequency value or frequency multiplier, depending by *iflag*. When *iflag* = 0, *xfreq* is a multiplier of a the default frequency, assigned by SF2 preset to the *inotenum* value. When *iflag* = 1, *xfreq* is the absolute frequency of the output sound, in Hz. Default is 0.

When *iflag* = 0, *inotenum* sets the frequency of the output according to the MIDI note number used, and *xfreq* is used as a multiplier. When *iflag* = 1, the frequency of the output, is set directly by *xfreq*. This allows the user to use any kind of micro-tuning based scales. However, this method is designed to work correctly only with presets tuned to the default equal temperament. Attempts to use this method with a preset already having non-standard tunings, or with drum-kit-based presets, could give unexpected results.

Adjustment of the amplitude can be done by varying the *xamp* argument, which acts as a multiplier.

Notice that both *xamp* and *xfreq* can use k-rate as well as a-rate signals. Both arguments must use variables of the same rate, or *sfplay* will not work correctly. *ipreindex* must contain the number of a previously assigned preset, or Csound will crash.

The *ioffset* parameter allows the sound to start from a sample different than the first one. The user should make sure that its value is within the length of the specific sound. Otherwise, Csound will probably crash.

The *ienv* parameter enables and determines the type of amplitude envelope used. The default value is 0, or no envelope. If *ienv* is set to 1, the attack and decay portions are linear. If set to 2, the attack is linear and the decay is exponential. The release portion of the envelope has not yet been implemented.

*sfplaym* is a mono version of *sfplay*. It should be used with mono preset, or with the stereo presets in which stereo output is not required. It is faster than *sfplay*.

These opcodes only support the sample structure of SF2 files. The modulator structure of the SoundFont2 format is not supported in Csound. Any modulation or processing to the sample data is left to the Csound user, bypassing all restrictions forced by the SF2 standard.

## Examples

Here is an example of the *sfplaym* opcode. It uses the file *sfplaym.csd* [examples/sfplaym.csd].

### Example 886. Example of the *sfplaym* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0 ;;realtime audio out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o sfplaym.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gisf sfload "07AcousticGuitar.sf2"
sfplist gisf
sfpassign 100, gisf

instr 1 ; play from score and midi keyboard

mididefault 60, p3
midinoteonkey p4, p5
inum init p4
ivel init p5
ivel init ivel/127 ;make velocity dependent
kamp linsegr 1, 1, 1, .1, 0
kamp = kamp/7000 ;scale amplitude
kfreq init 1 ;do not change freq from sf
aout sfplaym ivel, inum, kamp*ivel, kfreq, 100 ;preset index = 100
outs aout, aout

endin

</CsInstruments>
<CsScore>
f0 60 ; stay active for 1 minute

i1 0 1 60 127
i1 + 1 62 <
i1 + 1 65 <
```

```
i1 + 1 69 10
```

```
e  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*sfilist, sfinstr, sfinstrm, sfload, sfpassign, sfplay, sfplay3, sfplay3m, sfplist, sfpreset*

## Credits

Author: Gabriel Maldonado  
Italy  
May 2000

New in Csound Version 4.07

New optional parameter *ienv* in version 5.09

# sfplist

sfplist — Prints a list of all presets of a SoundFont2 (SF2) sample file.

## Description

Prints a list of all presets of a previously loaded SoundFont2 (SF2) sample file. These opcodes allow management the sample-structure of SF2 files. In order to understand the usage of these opcodes, the user must have some knowledge of the SF2 format, so a brief description of this format can be found in the *SoundFont2 File Format Appendix*.

## Syntax

```
sfplist ifilhandle
```

## Initialization

*ifilhandle* -- unique number generated by *sload* opcode to be used as an identifier for a SF2 file. Several SF2 files can be loaded and activated at the same time.

## Performance

*sfplist* prints a list of all presets of a previously loaded SF2 file to the console.

These opcodes only support the sample structure of SF2 files. The modulator structure of the SoundFont2 format is not supported in Csound. Any modulation or processing to the sample data is left to the Csound user, bypassing all restrictions forced by the SF2 standard.

## Examples

Here is an example of the sfplist opcode. It uses the file *sfplist.csd* [examples/sfplist.csd].

### Example 887. Example of the sfplist opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac --rtmidi=virtual -M0   ;;realtime audio out, virtual midi in
;-iadc      ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o sfplist.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gisf sfload "sf_GMbank.sf2"
```

```

sfplist gisf      ;list all presets
gir sfpreset 125, 3, gisf, 0      ;preset = Car Pass

instr 1 ; play from score and midi keyboard

  mididefault 60, p3
  midinoteonkey p4, p5
  inum init p4
  ivel init p5
  ivel init ivel/127      ;make velocity dependent
  kamp linsegr 1, 1, 1, .1, 0
  kamp = kamp/6000      ;scale amplitude
  kfreg init 1      ;do not change freq from sf
  a1,a2 sfplay3 ivel, inum, kamp, kfreg, gir
  outs a1, a2

endin

</CsInstruments>
<CsScore>
f0 60      ; stay active for 1 minute

i1 0 1 60 127
i1 + 1 62 <
i1 + 1 65 <
i1 + 1 69 10

e
</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like this:

```

Preset list of "sf_GMbank.sf2"
  0) Piano 1          prog:0   bank:0
  1) Piano 2          prog:1   bank:0
  2) Piano 3          prog:2   bank:0
  3) Honky Tonk      prog:3   bank:0
  4) E.Piano 1       prog:4   bank:0
  5) E.Piano 2       prog:5   bank:0
  6) Harpsichord     prog:6   bank:0
  .....
 146) Car-Pass       prog:125 bank:3
  .....

```

## See Also

*sfilist, sfinstr, sfinstrm, sfload, sfpassign, sfplay, sfplaym, sfpreset*

## Credits

Author: Gabriel Maldonado  
 Italy  
 May 2000

New in Csound Version 4.07

# sfpreset

`sfpreset` — Assigns an existing preset of a SoundFont2 (SF2) sample file to an index number.

## Description

Assigns an existing preset of a previously loaded SoundFont2 (SF2) sample file to an index number. These opcodes allow management the sample-structure of SF2 files. In order to understand the usage of these opcodes, the user must have some knowledge of the SF2 format, so a brief description of this format can be found in the *SoundFont2 File Format Appendix*.

*sfpreset* should be placed in the header section of a Csound orchestra.

## Syntax

```
ir sfpreset iprog, ibank, ifilhandle, ipreindex
```

## Initialization

*ir* -- output to be used by other SF2 opcodes. For *sfpreset*, *ir* is *ipreindex*.

*iprog* -- program number of a bank of presets in a SF2 file

*ibank* -- number of a specific bank of a SF2 file

*ifilhandle* -- unique number generated by *sfload* opcode to be used as an identifier for a SF2 file. Several SF2 files can be loaded and activated at the same time.

*ipreindex* -- preset index

## Performance

*sfpreset* assigns an existing preset of a previously loaded SF2 file to an index number, to be used later with the opcodes *sfplay* and *sfplaym*. The user must previously know the program and the bank numbers of the preset in order to fill the corresponding arguments. Any number of *sfpreset* instances can be placed in the header section of an orchestra, each one assigning a different preset belonging to the same (or different) SF2 file to different index numbers.

These opcodes only support the sample structure of SF2 files. The modulator structure of the SoundFont2 format is not supported in Csound. Any modulation or processing to the sample data is left to the Csound user, bypassing all restrictions forced by the SF2 standard.

## Examples

Here is an example of the `sfpreset` opcode. It uses the file *sfpreset.csd* [examples/sfpreset.csd].

### Example 888. Example of the `sfpreset` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0   ;;realtime audio out, virtual midi in
;-iadc   ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o sfpreset.wav -W   ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gisf1 sfload "sf_GMbank.sf2"
sfplist gisf1 ;list presets of first soundfont
gisf2 sfload "07AcousticGuitar.sf2"
sfplist gisf2 ;list presets of second soundfont
gir sfpreset 50, 0, gisf1, 0 ;assign Synth Strings to index 0
giv sfpreset 0, 0, gisf2, 1 ;assign AcousticGuitar to index 1
print gir
print giv

instr 1 ; play from score and midi keyboard

mididefault 60, p3
midinoteonkey p4, p5
inum init p4
ivel init p5
ivel init ivel/127 ;make velocity dependent
kamp linsegr 1, 1, 1, .1, 0
kamp = kamp/5000 ;scale amplitude
kfreq init 1 ;do not change freq from sf
a1,a2 sfplay3 ivel, inum, kamp*ivel, kfreq, p6
outs a1, a2

endin

</CsInstruments>
<CsScore>
f0 60 ; stay active for 1 minute

i1 0 1 60 127 0 ;= Synth Strings I from first soundfont
i1 + 1 62 < .
i1 + 1 65 < .
i1 + 1 69 10 .

i1 5 1 60 127 1 ;= AcousticGuitar from second soundfont
i1 + 1 62 < .
i1 + 1 65 < .
i1 + 1 69 10 .
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*sfilist, sfinstr, sfinstrm, sfload, sfpassign, sfplay, sfplaym, sfplist*

## Credits

Author: Gabriel Maldonado  
Italy

May 2000

New in Csound Version 4.07

# shaker

shaker — Sounds like the shaking of a maraca or similar gourd instrument.

## Description

Audio output is a tone related to the shaking of a maraca or similar gourd instrument. The method is a physically inspired model developed from Perry Cook, but re-coded for Csound.

## Syntax

```
ares shaker kamp, kfreq, kbeans, kdamp, ktimes [, idecay]
```

## Initialization

*idecay* -- If present indicates for how long at the end of the note the shaker is to be damped. The default value is zero.

## Performance

A note is played on a maraca-like instrument, with the arguments as below.

*kamp* -- Amplitude of note.

*kfreq* -- Frequency of note played.

*kbeans* -- The number of beans in the gourd. A value of 8 seems suitable.

*kdamp* -- The damping value of the shaker. Values of 0.98 to 1 seems suitable, with 0.99 a reasonable default.

*ktimes* -- Number of times shaken.



### Note

The argument *knum* was redundant, so it was removed in version 3.49.

## Examples

Here is an example of the shaker opcode. It uses the file *shaker.csd* [examples/shaker.csd].

### Example 889. Example of the shaker opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc      -d          ;;RT audio I/O
```

```
; For Non-realtime ouput leave only the line below:
; -o shaker.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1
instr 1
kfreq line p4, p3, 440
  a1 shaker 10000, kfreq, 8, 0.999, 100, 0
  out a1
endin

</CsInstruments>
<CsScore>

i 1 0 1 440
i 1 + 1 4000

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitc (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

Fixed the example thanks to a message from Istvan Varga.

# shiftin

shiftin — Shifts the contents of an audio variable into a 1-dimensional array.

## Description

This opcode can be used to push data from an audio variable into a 1-dimensional array. The array needs to be at least `ksmps` numbers long, but can be longer. Data is shifted in circularly, with the writing position moving by `ksmps` positions every `k-cycle`. When the array gets full, the writing position wraps around to the beginning of the array again (overwriting old positions). Together with the `shiftout` opcode, it can form a FIFO queue.

## Syntax

```
kout[] shiftin asig
```

## Performance

`kout[]` -- output array, needs to be at least `ksmps` numbers long.

`asig` -- input audio

## Examples

Here is an example of the `shiftin` opcode. It uses the file `shiftin.csd` [examples/shiftin.csd].

### Example 890. Example of the shiftin opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-d -odac
</CsOptions>
<CsInstruments>

instr 1
ideltime = 0.5
kDel[] init sr*0.5
a1 diskin2 "fox.wav",1,0,1
a2 shiftout kDel
kDel shiftin a1
      out a1 + a2
endin

</CsInstruments>
<CsScore>
i1 0 10
</CsScore>
</CsoundSynthesizer>
```

## See Also

Vectorial opcodes, array opcodes

## Credits

Author: Victor Lazzarini  
NUI Maynooth  
2014

New in version 6.04

# shiftout

shiftout — Shifts the contents of a 1-dimensional array into an audio variable.

## Description

This opcode can be used to push data to an audio variable from a 1-dimensional array. The array needs to be at least `ksmps` numbers long, but can be longer. Data is shifted out circularly, with the writing position moving by `ksmps` positions every `k-cycle`. When the array gets emptied, the writing position wraps around to the beginning of the array again. Together with the `shiftin` opcode, it can form a FIFO queue.

## Syntax

```
asig shiftout kIn[], ioff]
```

## Initialization

`ioff` -- initial read offset position (optional, defaults to 0).

## Performance

`kin[]` -- input array, needs to be at least `ksmps` numbers long.

`asig` -- output audio

## Examples

Here is an example of the `shiftout` opcode. It uses the file `shiftout.csd` [examples/shiftout.csd].

### Example 891. Example of the `shiftout` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-d -odac
</CsOptions>
<CsInstruments>

instr 1
ideltime = 0.5
kDel[] init sr*0.5
a1 diskin2 "fox.wav",1,0,1
a2 shiftout kDel
kDel shiftin a1
out a1 + a2
endin

</CsInstruments>
<CsScore>
i1 0 10
</CsScore>
</CsoundSynthesizer>
```

## See Also

Vectorial opcodes, array opcodes

## Credits

Author: Victor Lazzarini  
NUI Maynooth  
2014

New in version 6.04

# signum

signum — Performs a signum function.

## Description

Returns the signum of  $x$  returning -1, 0 or 1.

## Syntax

`signum(x)` (no rate restriction)

## Examples

Here is an example of the signum opcode. It uses the file `signum.csd` [examples/signum.csd].

### Example 892. Example of the signum opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o oscil.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr      = 44100
kr      = 4410
ksmps  = 10
nchnls = 2
0dbfs  = 1

gaArr[] init 2

instr 1
kEnv transeg 1, p3, -3, 0

a_pi = 4 * taninv(1.0);
a1   phasor 440;
a2   = sin(2 * a_pi * 1/ksmps * a1);
a3   dcblock2 a2
asig = signum(a3)

gaArr[0] = a2 * 0.6 * kEnv
gaArr[1] = asig * 0.6 * kEnv

outs gaArr[0], gaArr[1]
endin

</CsInstruments>
<CsScore>
i 1 0 3
```

```
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Author: John fitch  
University of Bath, Codemist Ltd.  
Bath, UK  
July 2013

New in Csound version 6.01

# sin

sin — Performs a sine function.

## Description

Returns the sine of  $x$  ( $x$  in radians).

## Syntax

`sin(x)` (no rate restriction)

`sin(k/i[])` (k- or i-arrays )

## Examples

Here is an example of the sin opcode. It uses the file `sin.csd` [examples/sin.csd].

### Example 893. Example of the sin opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o sin.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

isin1 =      sin(0)           ;sine of 0 is 0
isin2 =      sin($M_PI_2)     ;sine of pi/2 (1.5707...) is 1
isin3 =      sin($M_PI)       ;sine of pi (3.1415...) is 0
isin4 =      sin($M_PI_2 * 3) ;sine of 3/2pi (4.7123...) is -1
isin5 =      sin($M_PI * 2)    ;sine of 2pi (6.2831...) is 0
isin6 =      sin($M_PI * 4)    ;sine of 4pi is also 0
print      isin1, isin2, isin3, isin4, isin5, isin6

endin

instr 2 ;sin used in panning, after an example from Hans Mikelson

aout      vco2      0.8, 220      ; sawtooth
kpan      linseg    p4, p3, p5 ;0 = left, 1 = right
kpan      =         kpan*$M_PI_2   ;range 0-1 becomes 0-pi/2
kpanl     =         cos(kpan)
kpanr     =         sin(kpan)
outs      outs     aout*kpanl, aout*kpanr

endin
```

```
</CsInstruments>
<CsScore>
i 1 0 0
i 2 0 5 0 1 ;move left to right
i 2 5 5 1 0 ;move right to left
e
</CsScore>
</CsoundSynthesizer>
```

Its output should include a line like this:

```
instr 1: isin1 = 0.000 isin2 = 1.000 isin3 = 0.000 isin4 = -1.000 isin5 = -0.000 isin6 = 0.000
```

## See Also

*cos, cosh, cosinv, sinh, sininv, tan, tanh, taninv*

# sinh

sinh — Performs a hyperbolic sine function.

## Description

Returns the hyperbolic sine of  $x$  ( $x$  in radians).

## Syntax

```
sinh(x) (no rate  
restriction)  
sinh(k/i[]) (k- or i-arrays )
```

## Examples

Here is an example of the sinh opcode. It uses the file *sinh.csd* [examples/sinh.csd].

### Example 894. Example of the sinh opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
; Audio out  Audio in  
-odac          -iadc      ;;RT audio I/O  
; For Non-realtime ouput leave only the line below:  
; -o sinh.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>  
  
; Initialize the global variables.  
sr = 44100  
kr = 4410  
ksmps = 10  
nchnls = 1  
  
; Instrument #1.  
instr 1  
  irad = 1  
  i1 = sinh(irad)  
  
  print i1  
endin  
  
</CsInstruments>  
<CsScore>  
  
; Play Instrument #1 for one second.  
i 1 0 1  
e  
  
</CsScore>
```

```
</CsoundSynthesizer>
```

Its output should a line like this:

```
instr 1: i1 = 1.175
```

## See Also

*cos, cosh, cosinv, sin, sininv, tan, tanh, taninv*

## Credits

Example written by Kevin Conder.

New in version 3.47

# sininv

sininv — Performs an arcsine function.

## Description

Returns the arcsine of  $x$  ( $x$  in radians).

## Syntax

```
sininv(x) (no rate  
restriction)  
sininv(k/i[]) (k- or i-arrays)
```

## Examples

Here is an example of the sininv opcode. It uses the file *sininv.csd* [examples/sininv.csd].

### Example 895. Example of the sininv opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
; Audio out  Audio in  
-odac          -iadc      ;;RT audio I/O  
; For Non-realtime ouput leave only the line below:  
; -o sininv.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>  
  
; Initialize the global variables.  
sr = 44100  
kr = 4410  
ksmps = 10  
nchnls = 1  
  
; Instrument #1.  
instr 1  
  irad = 0.5  
  i1 = sininv(irad)  
  
  print i1  
endin  
  
</CsInstruments>  
<CsScore>  
  
; Play Instrument #1 for one second.  
i 1 0 1  
e  
  
</CsScore>
```

```
</CsoundSynthesizer>
```

Its output should include a line like this:

```
instr 1: i1 = 0.524
```

## See Also

*cos, cosh, cosinv, sin, sinh, tan, tanh, taninv*

## Credits

Author: John ffitch

New in version 3.48

Example written by Kevin Conder.

# sinsyn

sinsyn — Streaming partial track additive synthesis with cubic phase interpolation

## Description

The *sinsyn* opcode takes an input containing a TRACKS pv streaming signal (as generated, for instance by the *partials* opcode). It resynthesises the signal using linear amplitude and cubic phase interpolation to drive a bank of interpolating oscillators with amplitude scaling control. *sinsyn* attempts to preserve the phase of the partials in the original signal and in so doing it does not allow for pitch or timescale modifications of the signal.

## Syntax

```
asig sinsyn fin, kscal, kmaxtracks, ifn
```

## Performance

*asig* -- output audio rate signal

*fin* -- input pv stream in TRACKS format

*kscal* -- amplitude scaling

*kmaxtracks* -- max number of tracks in sinsynthesis. Limiting this will cause a non-linear filtering effect, by discarding newer and higher-frequency tracks (tracks are ordered by start time and ascending frequency, respectively)

*ifn* -- function table containing one cycle of a sinusoid (sine or cosine).

## Examples

Here is an example of the *sinsyn* opcode. It uses the file *sinsyn.csd* [examples/sinsyn.csd].

### Example 896. Example of the *sinsyn* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o sinsyn.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
```

```
instr 1

kxtr init p4
ain diskin2 "fox.wav", 1
fs1,fsi2 pvsifd ain, 2048, 512,1 ; ifd analysis
fst partials fs1, fsi2, .03, 1, 3, 500 ; partial tracking
aout sinsyn fst, .5, kxtr, 1 ; scale amplitude down
outs aout, aout

endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1

i 1 0 2.7 15 ;filtering effect by using low number of tracks
i 1 + 2.7 500 ;maximum number of tracks
e
</CsScore>
</CsSoundSynthesizer>
```

The example above shows partial tracking of an ifd-analysis signal and cubic-phase additive resynthesis.

## Credits

Author: Victor Lazzarini  
June 2005

New plugin in version 5

November 2004.

# sleighbells

sleighbells — Semi-physical model of a sleighbell sound.

## Description

*sleighbells* is a semi-physical model of a sleighbell sound. It is one of the PhISEM percussion opcodes. PhISEM (Physically Informed Stochastic Event Modeling) is an algorithmic approach for simulating collisions of multiple independent sound producing objects.

## Syntax

```
ares sleighbells kamp, idettack [, inum] [, idamp] [, imaxshake] [, ifreq] \  
    [, ifreq1] [, ifreq2]
```

## Initialization

*idettack* -- period of time over which all sound is stopped

*inum* (optional) -- The number of beads, teeth, bells, timbrels, etc. If zero, the default value is 32.

*idamp* (optional) -- the damping factor, as part of this equation:

$$\text{damping\_amount} = 0.9994 + (\text{idamp} * 0.002)$$

The default *damping\_amount* is 0.9994 which means that the default value of *idamp* is 0. The maximum *damping\_amount* is 1.0 (no damping). This means the maximum value for *idamp* is 0.03.

The recommended range for *idamp* is usually below 75% of the maximum value.

*imaxshake* (optional, default=0) -- amount of energy to add back into the system. The value should be in range 0 to 1.

*ifreq* (optional) -- the main resonant frequency. The default value is 2500.

*ifreq1* (optional) -- the first resonant frequency. The default value is 5300.

*ifreq2* (optional) -- the second resonant frequency. The default value is 6500.

## Performance

*kamp* -- Amplitude of output. Note: As these instruments are stochastic, this is only an approximation.

## Examples

Here is an example of the sleighbells opcode. It uses the file *sleighbells.csd* [examples/sleighbells.csd].

### Example 897. Example of the sleighbells opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o sleighbells.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

idamp = p4
asig sleighbells .7, 0.01, 32, idamp, 0.55
outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0.00 0.25 0 ;short sound
i 1 0.30 0.25
i 1 0.60 0.25
i 1 0.90 0.25
i 1 1.20 0.25
i 1 1.50 1 .3 ;longer sound
i 1 1.80 0.25 0 ;short sound again
i 1 2.10 0.25
i 1 2.40 0.25
i 1 2.70 0.25
i 1 3.00 0.25
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*bamboo, dripwater, guiro, tambourine*

## Credits

Author: Perry Cook, part of the PhISEM (Physically Informed Stochastic Event Modeling)  
Adapted by John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 4.07

Added notes by Rasmus Ekman on May 2002.

# slicearray

slicearray — Take a slice of a vector.

## Description

Take a slice of a vector (one-dimensional k-rate array).

## Syntax

```
karray slicearray kinarray, istart, iend
```

## Initialization

*istart* -- index of the first part of the answer.

*iend* -- index of the first element of the answer.

## Examples

Here is an example of the slicearray opcode. It uses the file *slicearray.csd* [examples/slicearray.csd].

### Example 898. Example of the slicearray opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

;create and fill an array
kArr[] genarray_i 1, 9

;print the content
printf "%s", 1, "kArr = whole array\n"
kndx = 0
until kndx == lenarray(kArr) do
printf "kArr[%d] = %f\n", kndx+1, kndx, kArr[kndx]
kndx += 1
od

;build new arrays for the slices
kArr1[] init 5
kArr2[] init 4
```

```
;put in first five and last four elements
kArr1  slicearray kArr, 0, 4
kArr2  slicearray kArr, 5, 8

;print the content
        printf "%s", 1, "\nkArr1 = slice from index 0 to index 4\n"
kndx   = 0
        until kndx == lenarray(kArr1) do
kndx   printf "kArr1[%d] = %f\n", kndx+1, kndx, kArr1[kndx]
        += 1
        od
        printf "%s", 1, "\nkArr2 = slice from index 5 to index 8\n"
kndx   = 0
        until kndx == lenarray(kArr2) do
kndx   printf "kArr2[%d] = %f\n", kndx+1, kndx, kArr2[kndx]
        += 1
        od

        turnoff
endin

</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsSoundSynthesizer>
```

## See Also

Vectorial opcodes

## Credits

Author: John ffitch  
Codemist Ltd  
2013

New in version 6.00

# slider16

`slider16` — Creates a bank of 16 different MIDI control message numbers.

## Description

Creates a bank of 16 different MIDI control message numbers.

## Syntax

```
i1,...,i16 slider16 ichan, ictlnum1, imin1, imax1, init1, ifn1,..., \  
    ictlnum16, imin16, imax16, init16, ifn16  
  
k1,...,k16 slider16 ichan, ictlnum1, imin1, imax1, init1, ifn1,..., \  
    ictlnum16, imin16, imax16, init16, ifn16
```

## Initialization

*i1 ... i16* -- output values

*ichan* -- MIDI channel (1-16)

*ictlnum1 ... ictlnum16* -- MIDI control number (0-127)

*imin1 ... imin16* -- minimum values for each controller

*imax1 ... imax16* -- maximum values for each controller

*init1 ... init16* -- initial value for each controller

*ifn1 ... ifn16* -- function table for conversion for each controller

## Performance

*k1 ... k16* -- output values

`slider16` is a bank of MIDI controllers, useful when using MIDI mixer such as Kawai MM-16 or others for changing whatever sound parameter in real-time. The raw MIDI control messages at the input port are converted to agree with *iminN* and *imaxN*, and an initial value can be set. Also, an optional non-interpolated function table with a custom translation curve is allowed, useful for enabling exponential response curves.

When no function table translation is required, set the *ifnN* value to 0, else set *ifnN* to a valid function table number. When table translation is enabled (i.e. setting *ifnN* value to a non-zero number referring to an already allocated function table), *initN* value should be set equal to *iminN* or *imaxN* value, else the initial output value will not be the same as specified in *initN* argument.

`slider16` allows a bank of 16 different MIDI control message numbers.

As the input and output arguments are many, you can split the line using `\` (backslash) character (new in 3.47 version) to improve the readability. Using these opcodes is considerably more efficient than using the separate ones (`ctrl7` and `tonek`) when more controllers are required.

In the i-rate version of `slider16`, there is not an initial value input argument, because the output is gotten directly from current status of internal controller array of Csound.

## See Also

*s16b14, s32b14, slider16f, slider32, slider32f, slider64, slider64f, slider8, slider8f*

## Credits

Author: Gabriel Maldonado  
Italy  
December 1998

New in Csound version 3.50

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# slider16f

slider16f — Creates a bank of 16 different MIDI control message numbers, filtered before output.

## Description

Creates a bank of 16 different MIDI control message numbers, filtered before output.

## Syntax

```
k1,...,k16 slider16f ichan, ictlnum1, imin1, imax1, init1, ifn1, \  
    icutoff1,..., ictlnum16, imin16, imax16, init16, ifn16, icutoff16
```

## Initialization

*ichan* -- MIDI channel (1-16)

*ictlnum1* ... *ictlnum16* -- MIDI control number (0-127)

*imin1* ... *imin16* -- minimum values for each controller

*imax1* ... *imax16* -- maximum values for each controller

*init1* ... *init16* -- initial value for each controller

*ifn1* ... *ifn16* -- function table for conversion for each controller

*icutoff1* ... *icutoff16* -- low-pass filter cutoff frequency for each controller

## Performance

*k1* ... *k16* -- output values

*slider16f* is a bank of MIDI controllers, useful when using MIDI mixer such as Kawai MM-16 or others for changing whatever sound parameter in real-time. The raw MIDI control messages at the input port are converted to agree with *iminN* and *imaxN*, and an initial value can be set. Also, an optional non-interpolated function table with a custom translation curve is allowed, useful for enabling exponential response curves.

When no function table translation is required, set the *ifnN* value to 0, else set *ifnN* to a valid function table number. When table translation is enabled (i.e. setting *ifnN* value to a non-zero number referring to an already allocated function table), *initN* value should be set equal to *iminN* or *imaxN* value, else the initial output value will not be the same as specified in *initN* argument.

*slider16f* allows a bank of 16 different MIDI control message numbers. It filters the signal before output. This eliminates discontinuities due to the low resolution of the MIDI (7 bit). The cutoff frequency can be set separately for each controller (suggested range: .1 to 5 Hz).

As the input and output arguments are many, you can split the line using '\ (backslash) character (new in 3.47 version) to improve the readability. Using these opcodes is considerably more efficient than using the separate ones (*ctrl7* and *tonek*) when more controllers are required.



## Warning

*slider16f* does not output the required initial value immediately, but only after some k-cycles because the filter slightly delays the output.

## See Also

*s16b14*, *s32b14*, *slider16*, *slider32*, *slider32f*, *slider64*, *slider64f*, *slider8*, *slider8f*

## Credits

Author: Gabriel Maldonado  
Italy  
December 1998

New in Csound version 3.50

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# slider16table

slider16table — Stores a bank of 16 different MIDI control messages to a table.

## Description

Stores a bank of 16 different MIDI control messages to a table.

## Syntax

```
kflag slider16table ichan, ioutTable, ioffset, ictlnum1, imin1, imax1, \  
    init1, ifn1, ... , ictlnum16, imin16, imax16, init16, ifn16
```

## Initialization

*ichan* -- MIDI channel (1-16)

*ioutTable* -- number of the table that will contain the output

*ioffset* -- output table offset. A zero means that the output of the first slider will affect the first table element. A 10 means that the output of the first slider will affect the 11th table element.

*ictlnum1* ... *ictlnum16* -- MIDI control number (0-127)

*imin1* ... *imin16* -- minimum values for each controller

*imax1* ... *imax16* -- maximum values for each controller

*init1* ... *init16* -- initial value for each controller

*ifn1* ... *ifn16* -- function table for conversion for each controller

## Performance

*kflag* -- a flag that informs if any control-change message in the bank has been received. In this case *kflag* is set to 1. Otherwise is set to zero.

*slider16table* is a bank of MIDI controllers, useful when using MIDI mixer such as Kawai MM-16 or others for changing whatever sound parameter in real-time. The raw MIDI control messages at the input port are converted to agree with *iminN* and *imaxN*, and an initial value can be set. Also, an optional non-interpolated function table with a custom translation curve is allowed, useful for enabling exponential response curves.

When no function table translation is required, set the *ifnN* value to 0, else set *ifnN* to a valid function table number. When table translation is enabled (i.e. setting *ifnN* value to a non-zero number referring to an already allocated function table), *initN* value should be set equal to *iminN* or *imaxN* value, else the initial output value will not be the same as specified in *initN* argument.

*slider16table* allows a bank of 16 different MIDI control message numbers.

As the input and output arguments are many, you can split the line using `\` (backslash) character (new in 3.47 version) to improve the readability. Using these opcodes is considerably more efficient than using the separate ones (*ctrl7* and *tonek*) when more controllers are required.

*slider16table* is very similar to *slider16* and *sliderN* family of opcodes (see their manual for more information). The actual difference is that the output is not stored to k-rate variables, but to a table, denoted by the *ioutTable* argument. It is possible to define a starting index in order to use the same table for more than one slider bank (or other purposes).

It is possible to use this opcode together with *FLslidBnk2Setk* and *FLslidBnk2*, so you can synchronize the position of the MIDI values to the position of the FLTK valuator widgets of *FLslidBnk2*. Notice that you have to specify the same min/max values as well the linear/exponential responses in both *sliderNtable(f)* and *FLslidBnk2*. The exception is when using table-indexed response instead of a lin/exp response. In this case, in order to achieve a useful result, the table-indexed response and actual min/max values must be set only in *FLslidBnk2*, whereas, in *sliderNtable(f)*, you have to set a linear response and a minimum of zero and a maximum of one in all sliders.

## See Also

*slider16tablef*, *slider32table*, *slider32tablef*, *slider64table*, *slider64tablef*, *slidertable8*, *slider8tablef*

## Credits

Author: Gabriel Maldonado

New in Csound version 5.06

# slider16tablef

slider16tablef — Stores a bank of 16 different MIDI control messages to a table, filtered before output.

## Description

Stores a bank of 16 different MIDI control messages to a table, filtered before output.

## Syntax

```
kflag slider16tablef ichan, ioutTable, ioffset, ictlnum1, imin1, imax1, \  
    init1, ifn1, icutoff1, ..., ictlnum16, imin16, imax16, init16, ifn16, icutoff16
```

## Initialization

*ichan* -- MIDI channel (1-16)

*ioutTable* -- number of the table that will contain the output

*ioffset* -- output table offset. A zero means that the output of the first slider will affect the first table element. A 10 means that the output of the first slider will affect the 11th table element.

*ictlnum1* ... *ictlnum16* -- MIDI control number (0-127)

*imin1* ... *imin16* -- minimum values for each controller

*imax1* ... *imax16* -- maximum values for each controller

*init1* ... *init16* -- initial value for each controller

*ifn1* ... *ifn16* -- function table for conversion for each controller

*icutoff1* ... *icutoff16* -- low-pass filter cutoff frequency for each controller

## Performance

*kflag* -- a flag that informs if any control-change message in the bank has been received. In this case *kflag* is set to 1. Otherwise is set to zero.

*slider16tablef* is a bank of MIDI controllers, useful when using MIDI mixer such as Kawai MM-16 or others for changing whatever sound parameter in real-time. The raw MIDI control messages at the input port are converted to agree with *iminN* and *imaxN*, and an initial value can be set. Also, an optional non-interpolated function table with a custom translation curve is allowed, useful for enabling exponential response curves.

When no function table translation is required, set the *ifnN* value to 0, else set *ifnN* to a valid function table number. When table translation is enabled (i.e. setting *ifnN* value to a non-zero number referring to an already allocated function table), *initN* value should be set equal to *iminN* or *imaxN* value, else the initial output value will not be the same as specified in *initN* argument.

*slider16tablef* allows a bank of 16 different MIDI control message numbers. It filters the signal before output. This eliminates discontinuities due to the low resolution of the MIDI (7 bit). The cutoff frequency can be set separately for each controller (suggested range: .1 to 5 Hz).

As the input and output arguments are many, you can split the line using `\` (backslash) character (new in 3.47 version) to improve the readability. Using these opcodes is considerably more efficient than using the separate ones (*ctrl7* and *tonek*) when more controllers are required.

*slider16tablef* is very similar to *slider16f* and *sliderNf* family of opcodes (see their manual for more information). The actual difference is that the output is not stored to k-rate variables, but to a table, denoted by the *ioutTable* argument. It is possible to define a starting index in order to use the same table for more than one slider bank (or other purposes).

It is possible to use this opcode together with *FLslidBnk2Setk* and *FLslidBnk2*, so you can synchronize the position of the MIDI values to the position of the FLTK valuator widgets of *FLslidBnk2*. Notice that you have to specify the same min/max values as well the linear/exponential responses in both *sliderNtable(f)* and *FLslidBnk2*. The exception is when using table-indexed response instead of a lin/exp response. In this case, in order to achieve a useful result, the table-indexed response and actual min/max values must be set only in *FLslidBnk2*, whereas, in *sliderNtable(f)*, you have to set a linear response and a minimum of zero and a maximum of one in all sliders.



### Warning

*slider16tablef* does not output the required initial value immediately, but only after some k-cycles because the filter slightly delays the output.

## See Also

*slider16table*, *slider32table*, *slider32tablef*, *slider64table*, *slider64tablef*, *slider8table*, *slider8tablef*

## Credits

Author: Gabriel Maldonado

New in Csound version 5.06

# slider32

`slider32` — Creates a bank of 32 different MIDI control message numbers.

## Description

Creates a bank of 32 different MIDI control message numbers.

## Syntax

```
i1,...,i32 slider32 ichan, ictlnum1, imin1, imax1, init1, ifn1,..., \  
    ictlnum32, imin32, imax32, init32, ifn32  
  
k1,...,k32 slider32 ichan, ictlnum1, imin1, imax1, init1, ifn1,..., \  
    ictlnum32, imin32, imax32, init32, ifn32
```

## Initialization

*i1* ... *i32* -- output values

*ichan* -- MIDI channel (1-16)

*ictlnum1* ... *ictlnum32* -- MIDI control number (0-127)

*imin1* ... *imin32* -- minimum values for each controller

*imax1* ... *imax32* -- maximum values for each controller

*init1* ... *init32* -- initial value for each controller

*ifn1* ... *ifn32* -- function table for conversion for each controller

## Performance

*k1* ... *k32* -- output values

`slider32` is a bank of MIDI controllers, useful when using MIDI mixer such as Kawai MM-16 or others for changing whatever sound parameter in real-time. The raw MIDI control messages at the input port are converted to agree with *iminN* and *imaxN*, and an initial value can be set. Also, an optional non-interpolated function table with a custom translation curve is allowed, useful for enabling exponential response curves.

When no function table translation is required, set the *ifnN* value to 0, else set *ifnN* to a valid function table number. When table translation is enabled (i.e. setting *ifnN* value to a non-zero number referring to an already allocated function table), *initN* value should be set equal to *iminN* or *imaxN* value, else the initial output value will not be the same as specified in *initN* argument.

`slider32` allows a bank of 32 different MIDI control message numbers.

As the input and output arguments are many, you can split the line using `\` (backslash) character (new in 3.47 version) to improve the readability. Using these opcodes is considerably more efficient than using the separate ones (`ctrl7` and `tonek`) when more controllers are required.

In the i-rate version of `slider32`, there is not an initial value input argument, because the output is gotten directly from current status of internal controller array of Csound.

## See Also

*s16b14, s32b14, slider16, slider16f, slider32f, slider64, slider64f, slider8, slider8f*

## Credits

Author: Gabriel Maldonado  
Italy  
December 1998

New in Csound version 3.50

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# slider32f

slider32f — Creates a bank of 32 different MIDI control message numbers, filtered before output.

## Description

Creates a bank of 32 different MIDI control message numbers, filtered before output.

## Syntax

```
k1,...,k32 slider32f ichan, ictlnum1, imin1, imax1, init1, ifn1, icutoff1, \  
..., ictlnum32, imin32, imax32, init32, ifn32, icutoff32
```

## Initialization

*ichan* -- MIDI channel (1-16)

*ictlnum1* ... *ictlnum32* -- MIDI control number (0-127)

*imin1* ... *imin32* -- minimum values for each controller

*imax1* ... *imax32* -- maximum values for each controller

*init1* ... *init32* -- initial value for each controller

*ifn1* ... *ifn32* -- function table for conversion for each controller

*icutoff1* ... *icutoff32* -- low-pass filter cutoff frequency for each controller

## Performance

*k1* ... *k32* -- output values

*slider32f* is a bank of MIDI controllers, useful when using MIDI mixer such as Kawai MM-16 or others for changing whatever sound parameter in real-time. The raw MIDI control messages at the input port are converted to agree with *iminN* and *imaxN*, and an initial value can be set. Also, an optional non-interpolated function table with a custom translation curve is allowed, useful for enabling exponential response curves.

When no function table translation is required, set the *ifnN* value to 0, else set *ifnN* to a valid function table number. When table translation is enabled (i.e. setting *ifnN* value to a non-zero number referring to an already allocated function table), *initN* value should be set equal to *iminN* or *imaxN* value, else the initial output value will not be the same as specified in *initN* argument.

*slider32f* allows a bank of 32 different MIDI control message numbers. It filters the signal before output. This eliminates discontinuities due to the low resolution of the MIDI (7 bit). The cutoff frequency can be set separately for each controller (suggested range: .1 to 5 Hz).

As the input and output arguments are many, you can split the line using `\` (backslash) character (new in 3.47 version) to improve the readability. Using these opcodes is considerably more efficient than using the separate ones (*ctrl7* and *tonek*) when more controllers are required.



## Warning

*slider32f* opcodes do not output the required initial value immediately, but only after some k-cycles because the filter slightly delays the output.

## See Also

*s16b14, s32b14, slider16, slider16f, slider32, slider64, slider64f, slider8, slider8f*

## Credits

Author: Gabriel Maldonado  
Italy  
December 1998

New in Csound version 3.50

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# slider32table

slider32table — Stores a bank of 32 different MIDI control messages to a table.

## Description

Creates a bank of 32 different MIDI control messages to a table.

## Syntax

```
kflag slider32table ichan, ioutTable, ioffset, ictlnum1, imin1, \  
imax1, init1, ifn1, ... , ictlnum32, imin32, imax32, init32, ifn32
```

## Initialization

*ichan* -- MIDI channel (1-16)

*ioutTable* -- number of the table that will contain the output

*ioffset* -- output table offset. A zero means that the output of the first slider will affect the first table element. A 10 means that the output of the first slider will affect the 11th table element.

*ictlnum1* ... *ictlnum32* -- MIDI control number (0-127)

*imin1* ... *imin32* -- minimum values for each controller

*imax1* ... *imax32* -- maximum values for each controller

*init1* ... *init32* -- initial value for each controller

*ifn1* ... *ifn32* -- function table for conversion for each controller

## Performance

*kflag* -- a flag that informs if any control-change message in the bank has been received. In this case *kflag* is set to 1. Otherwise is set to zero.

*slider32table* is a bank of MIDI controllers, useful when using MIDI mixer such as Kawai MM-16 or others for changing whatever sound parameter in real-time. The raw MIDI control messages at the input port are converted to agree with *iminN* and *imaxN*, and an initial value can be set. Also, an optional non-interpolated function table with a custom translation curve is allowed, useful for enabling exponential response curves.

When no function table translation is required, set the *ifnN* value to 0, else set *ifnN* to a valid function table number. When table translation is enabled (i.e. setting *ifnN* value to a non-zero number referring to an already allocated function table), *initN* value should be set equal to *iminN* or *imaxN* value, else the initial output value will not be the same as specified in *initN* argument.

*slider32table* allows a bank of 32 different MIDI control message numbers.

As the input and output arguments are many, you can split the line using `\` (backslash) character (new in 3.47 version) to improve the readability. Using these opcodes is considerably more efficient than using the separate ones (*ctrl7* and *tonek*) when more controllers are required.

*slider32table* is very similar to *slider32* and *sliderN* family of opcodes (see their manual for more information). The actual difference is that the output is not stored to k-rate variables, but to a table, denoted by the *ioutTable* argument. It is possible to define a starting index in order to use the same table for more than one slider bank (or other purposes).

It is possible to use this opcode together with *FLslidBnk2Setk* and *FLslidBnk2*, so you can synchronize the position of the MIDI values to the position of the FLTK valuator widgets of *FLslidBnk2*. Notice that you have to specify the same min/max values as well the linear/exponential responses in both *sliderNtable(f)* and *FLslidBnk2*. The exception is when using table-indexed response instead of a lin/exp response. In this case, in order to achieve a useful result, the table-indexed response and actual min/max values must be set only in *FLslidBnk2*, whereas, in *sliderNtable(f)*, you have to set a linear response and a minimum of zero and a maximum of one in all sliders.

## See Also

*slider16table*, *slider16tablef*, *slider32tablef*, *slider64table*, *slider64tablef*, *slider8table*, *slider8tablef*

## Credits

Author: Gabriel Maldonado

New in Csound version 5.06

# slider32tablef

`slider32tablef` — Stores a bank of 32 different MIDI control messages to a table, filtered before output.

## Description

Stores a bank of 32 different MIDI control messages to a table, filtered before output.

## Syntax

```
kflag slider32tablef ichan, ioutTable, ioffset, ictlnum1, imin1, imax1, \  
init1, ifn1, icutoff1, ..., ictlnum32, imin32, imax32, init32, ifn32, icutoff32
```

## Initialization

*ichan* -- MIDI channel (1-16)

*ioutTable* -- number of the table that will contain the output

*ioffset* -- output table offset. A zero means that the output of the first slider will affect the first table element. A 10 means that the output of the first slider will affect the 11th table element.

*ictlnum1* ... *ictlnum32* -- MIDI control number (0-127)

*imin1* ... *imin32* -- minimum values for each controller

*imax1* ... *imax32* -- maximum values for each controller

*init1* ... *init32* -- initial value for each controller

*ifn1* ... *ifn32* -- function table for conversion for each controller

*icutoff1* ... *icutoff32* -- low-pass filter cutoff frequency for each controller

## Performance

*kflag* -- a flag that informs if any control-change message in the bank has been received. In this case *kflag* is set to 1. Otherwise is set to zero.

`slider32tablef` is a bank of MIDI controllers, useful when using MIDI mixer such as Kawai MM-16 or others for changing whatever sound parameter in real-time. The raw MIDI control messages at the input port are converted to agree with *iminN* and *imaxN*, and an initial value can be set. Also, an optional non-interpolated function table with a custom translation curve is allowed, useful for enabling exponential response curves.

When no function table translation is required, set the *ifnN* value to 0, else set *ifnN* to a valid function table number. When table translation is enabled (i.e. setting *ifnN* value to a non-zero number referring to an already allocated function table), *initN* value should be set equal to *iminN* or *imaxN* value, else the initial output value will not be the same as specified in *initN* argument.

`slider32tablef` allows a bank of 32 different MIDI control message numbers. It filters the signal before output. This eliminates discontinuities due to the low resolution of the MIDI (7 bit). The cutoff frequency can be set separately for each controller (suggested range: .1 to 5 Hz).

As the input and output arguments are many, you can split the line using `\` (backslash) character (new in 3.47 version) to improve the readability. Using these opcodes is considerably more efficient than using the separate ones (*ctrl7* and *tonek*) when more controllers are required.

*slider32tablef* is very similar to *slider32f* and *sliderNf* family of opcodes (see their manual for more information). The actual difference is that the output is not stored to k-rate variables, but to a table, denoted by the *ioutTable* argument. It is possible to define a starting index in order to use the same table for more than one slider bank (or other purposes).

It is possible to use this opcode together with *FLslidBnk2Setk* and *FLslidBnk2*, so you can synchronize the position of the MIDI values to the position of the FLTK valuator widgets of *FLslidBnk2*. Notice that you have to specify the same min/max values as well the linear/exponential responses in both *sliderNtable(f)* and *FLslidBnk2*. The exception is when using table-indexed response instead of a lin/exp response. In this case, in order to achieve a useful result, the table-indexed response and actual min/max values must be set only in *FLslidBnk2*, whereas, in *sliderNtable(f)*, you have to set a linear response and a minimum of zero and a maximum of one in all sliders.



### Warning

*slider32tablef* opcodes do not output the required initial value immediately, but only after some k-cycles because the filter slightly delays the output.

## See Also

*slider16*, *slider16f*, *slider32*, *slider64*, *slider64f*, *slider8*, *slider8f*

## Credits

Author: Gabriel Maldonado

New in Csound version 5.06

# slider64

slider64 — Creates a bank of 64 different MIDI control message numbers.

## Description

Creates a bank of 64 different MIDI control message numbers.

## Syntax

```
i1,...,i64 slider64 ichan, ictlnum1, imin1, imax1, init1, ifn1,..., \  
    ictlnum64, imin64, imax64, init64, ifn64  
  
k1,...,k64 slider64 ichan, ictlnum1, imin1, imax1, init1, ifn1,..., \  
    ictlnum64, imin64, imax64, init64, ifn64
```

## Initialization

*i1 ... i64* -- output values

*ichan* -- MIDI channel (1-16)

*ictlnum1 ... ictlnum64* -- MIDI control number (0-127)

*imin1 ... imin64* -- minimum values for each controller

*imax1 ... imax64* -- maximum values for each controller

*init1 ... init64* -- initial value for each controller

*ifn1 ... ifn64* -- function table for conversion for each controller

## Performance

*k1 ... k64* -- output values

*slider64* is a bank of MIDI controllers, useful when using MIDI mixer such as Kawai MM-16 or others for changing whatever sound parameter in real-time. The raw MIDI control messages at the input port are converted to agree with *iminN* and *imaxN*, and an initial value can be set. Also, an optional non-interpolated function table with a custom translation curve is allowed, useful for enabling exponential response curves.

When no function table translation is required, set the *ifnN* value to 0, else set *ifnN* to a valid function table number. When table translation is enabled (i.e. setting *ifnN* value to a non-zero number referring to an already allocated function table), *initN* value should be set equal to *iminN* or *imaxN* value, else the initial output value will not be the same as specified in *initN* argument.

*slider64* allows a bank of 64 different MIDI control message numbers.

As the input and output arguments are many, you can split the line using `\` (backslash) character (new in 3.47 version) to improve the readability. Using these opcodes is considerably more efficient than using the separate ones (*ctrl7* and *tonek*) when more controllers are required.

In the i-rate version of *slider64*, there is not an initial value input argument, because the output is gotten directly from current status of internal controller array of Csound.

## See Also

*s16b14, s32b14, slider16, slider16f, slider32, slider32f, slider64f slider8, slider8f*

## Credits

Author: Gabriel Maldonado  
Italy  
December 1998

New in Csound version 3.50

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# slider64f

slider64f — Creates a bank of 64 different MIDI control message numbers, filtered before output.

## Description

Creates a bank of 64 different MIDI control message numbers, filtered before output.

## Syntax

```
k1,...,k64 slider64f ichan, ictlnum1, imin1, imax1, init1, ifn1, \  
    icutoff1,..., ictlnum64, imin64, imax64, init64, ifn64, icutoff64
```

## Initialization

*ichan* -- MIDI channel (1-16)

*ictlnum1* ... *ictlnum64* -- MIDI control number (0-127)

*imin1* ... *imin64* -- minimum values for each controller

*imax1* ... *imax64* -- maximum values for each controller

*init1* ... *init64* -- initial value for each controller

*ifn1* ... *ifn64* -- function table for conversion for each controller

*icutoff1* ... *icutoff64* -- low-pass filter cutoff frequency for each controller

## Performance

*k1* ... *k64* -- output values

*slider64f* is a bank of MIDI controllers, useful when using MIDI mixer such as Kawai MM-16 or others for changing whatever sound parameter in real-time. The raw MIDI control messages at the input port are converted to agree with *iminN* and *imaxN*, and an initial value can be set. Also, an optional non-interpolated function table with a custom translation curve is allowed, useful for enabling exponential response curves.

When no function table translation is required, set the *ifnN* value to 0, else set *ifnN* to a valid function table number. When table translation is enabled (i.e. setting *ifnN* value to a non-zero number referring to an already allocated function table), *initN* value should be set equal to *iminN* or *imaxN* value, else the initial output value will not be the same as specified in *initN* argument.

*slider64f* allows a bank of 64 different MIDI control message numbers. It filters the signal before output. This eliminates discontinuities due to the low resolution of the MIDI (7 bit). The cutoff frequency can be set separately for each controller (suggested range: .1 to 5 Hz).

As the input and output arguments are many, you can split the line using `\` (backslash) character (new in 3.47 version) to improve the readability. Using these opcodes is considerably more efficient than using the separate ones (*ctrl7* and *tonek*) when more controllers are required.



## Warning

*slider64f* opcodes do not output the required initial value immediately, but only after some k-cycles because the filter slightly delays the output.

## See Also

*s16b14, s32b14, slider16, slider16f, slider32, slider32f, slider64, slider8, slider8f*

## Credits

Author: Gabriel Maldonado  
Italy  
December 1998

New in Csound version 3.50

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# slider64table

slider64table — Stores a bank of 64 different MIDI control messages to a table.

## Description

Creates a bank of 64 different MIDI control messages to a table.

## Syntax

```
kflag slider64table ichan, ioutTable, ioffset, ictlnum1, imin1, \  
imax1, init1, ifn1, ... , ictlnum64, imin64, imax64, init64, ifn64
```

## Initialization

*ichan* -- MIDI channel (1-16)

*ioutTable* -- number of the table that will contain the output

*ioffset* -- output table offset. A zero means that the output of the first slider will affect the first table element. A 10 means that the output of the first slider will affect the 11th table element.

*ictlnum1* ... *ictlnum64* -- MIDI control number (0-127)

*imin1* ... *imin64* -- minimum values for each controller

*imax1* ... *imax64* -- maximum values for each controller

*init1* ... *init64* -- initial value for each controller

*ifn1* ... *ifn64* -- function table for conversion for each controller

## Performance

*kflag* -- a flag that informs if any control-change message in the bank has been received. In this case *kflag* is set to 1. Otherwise is set to zero.

*slider64table* is a bank of MIDI controllers, useful when using MIDI mixer such as Kawai MM-16 or others for changing whatever sound parameter in real-time. The raw MIDI control messages at the input port are converted to agree with *iminN* and *imaxN*, and an initial value can be set. Also, an optional non-interpolated function table with a custom translation curve is allowed, useful for enabling exponential response curves.

When no function table translation is required, set the *ifnN* value to 0, else set *ifnN* to a valid function table number. When table translation is enabled (i.e. setting *ifnN* value to a non-zero number referring to an already allocated function table), *initN* value should be set equal to *iminN* or *imaxN* value, else the initial output value will not be the same as specified in *initN* argument.

*slider64table* allows a bank of 64 different MIDI control message numbers.

As the input and output arguments are many, you can split the line using `\` (backslash) character (new in 3.47 version) to improve the readability. Using these opcodes is considerably more efficient than using the separate ones (*ctrl7* and *tonek*) when more controllers are required.

*slider64table* is very similar to *slider64* and *sliderN* family of opcodes (see their manual for more information). The actual difference is that the output is not stored to k-rate variables, but to a table, denoted by the *ioutTable* argument. It is possible to define a starting index in order to use the same table for more than one slider bank (or other purposes).

It is possible to use this opcode together with *FLslidBnk2Setk* and *FLslidBnk2*, so you can synchronize the position of the MIDI values to the position of the FLTK valuator widgets of *FLslidBnk2*. Notice that you have to specify the same min/max values as well the linear/exponential responses in both *sliderNtable(f)* and *FLslidBnk2*. The exception is when using table-indexed response instead of a lin/exp response. In this case, in order to achieve a useful result, the table-indexed response and actual min/max values must be set only in *FLslidBnk2*, whereas, in *sliderNtable(f)*, you have to set a linear response and a minimum of zero and a maximum of one in all sliders.

## See Also

*slider16table*, *slider16tablef*, *slider32table*, *slider32tablef*, *slider64tablef*, *slider8table*, *slider8tablef*

## Credits

Author: Gabriel Maldonado

New in Csound version 5.06

# slider64tablef

slider64tablef — Stores a bank of 64 different MIDI control messages to a table, filtered before output.

## Description

Stores a bank of 64 different MIDI MIDI control messages to a table, filtered before output.

## Syntax

```
kflag slider64tablef ichan, ioutTable, ioffset, ictlnum1, imin1, imax1, \  
    init1, ifn1, icutoff1, ..., ictlnum64, imin64, imax64, init64, ifn64, icutoff64
```

## Initialization

*ichan* -- MIDI channel (1-16)

*ioutTable* -- number of the table that will contain the output

*ioffset* -- output table offset. A zero means that the output of the first slider will affect the first table element. A 10 means that the output of the first slider will affect the 11th table element.

*ictlnum1* ... *ictlnum64* -- MIDI control number (0-127)

*imin1* ... *imin64* -- minimum values for each controller

*imax1* ... *imax64* -- maximum values for each controller

*init1* ... *init64* -- initial value for each controller

*ifn1* ... *ifn64* -- function table for conversion for each controller

*icutoff1* ... *icutoff64* -- low-pass filter cutoff frequency for each controller

## Performance

*kflag* -- a flag that informs if any control-change message in the bank has been received. In this case *kflag* is set to 1. Otherwise is set to zero.

*slider64tablef* is a bank of MIDI controllers, useful when using MIDI mixer such as Kawai MM-16 or others for changing whatever sound parameter in real-time. The raw MIDI control messages at the input port are converted to agree with *iminN* and *imaxN*, and an initial value can be set. Also, an optional non-interpolated function table with a custom translation curve is allowed, useful for enabling exponential response curves.

When no function table translation is required, set the *ifnN* value to 0, else set *ifnN* to a valid function table number. When table translation is enabled (i.e. setting *ifnN* value to a non-zero number referring to an already allocated function table), *initN* value should be set equal to *iminN* or *imaxN* value, else the initial output value will not be the same as specified in *initN* argument.

*slider64tablef* allows a bank of 64 different MIDI control message numbers. It filters the signal before output. This eliminates discontinuities due to the low resolution of the MIDI (7 bit). The cutoff frequency can be set separately for each controller (suggested range: .1 to 5 Hz).

As the input and output arguments are many, you can split the line using `\` (backslash) character (new in 3.47 version) to improve the readability. Using these opcodes is considerably more efficient than using the separate ones (*ctrl7* and *tonek*) when more controllers are required.

*slider64tablef* is very similar to *slider64f* and *sliderN* family of opcodes (see their manual for more information). The actual difference is that the output is not stored to k-rate variables, but to a table, denoted by the *ioutTable* argument. It is possible to define a starting index in order to use the same table for more than one slider bank (or other purposes).

It is possible to use this opcode together with *FLslidBnk2Setk* and *FLslidBnk2*, so you can synchronize the position of the MIDI values to the position of the FLTK valuator widgets of *FLslidBnk2*. Notice that you have to specify the same min/max values as well the linear/exponential responses in both *sliderNtable(f)* and *FLslidBnk2*. The exception is when using table-indexed response instead of a lin/exp response. In this case, in order to achieve a useful result, the table-indexed response and actual min/max values must be set only in *FLslidBnk2*, whereas, in *sliderNtable(f)*, you have to set a linear response and a minimum of zero and a maximum of one in all sliders.



### Warning

*slider64tablef* opcodes do not output the required initial value immediately, but only after some k-cycles because the filter slightly delays the output.

## See Also

*slider16table*, *slider16tablef*, *slider32table*, *slider32tablef*, *slider64table*, *slider8table*, *slider8tablef*

## Credits

Author: Gabriel Maldonado

New in Csound version 5.06

# slider8

slider8 — Creates a bank of 8 different MIDI control message numbers.

## Description

Creates a bank of 8 different MIDI control message numbers.

## Syntax

```
i1,...,i8 slider8 ichan, ictlnum1, imin1, imax1, init1, ifn1,..., \  
    ictlnum8, imin8, imax8, init8, ifn8  
  
k1,...,k8 slider8 ichan, ictlnum1, imin1, imax1, init1, ifn1,..., \  
    ictlnum8, imin8, imax8, init8, ifn8
```

## Initialization

*i1 ... i8* -- output values

*ichan* -- MIDI channel (1-16)

*ictlnum1 ... ictlnum8* -- MIDI control number (0-127)

*imin1 ... imin8* -- minimum values for each controller

*imax1 ... imax8* -- maximum values for each controller

*init1 ... init8* -- initial value for each controller

*ifn1 ... ifn8* -- function table for conversion for each controller

## Performance

*k1 ... k8* -- output values

*slider8* is a bank of MIDI controllers, useful when using MIDI mixer such as Kawai MM-16 or others for changing whatever sound parameter in real-time. The raw MIDI control messages at the input port are converted to agree with *iminN* and *imaxN*, and an initial value can be set. Also, an optional non-interpolated function table with a custom translation curve is allowed, useful for enabling exponential response curves.

When no function table translation is required, set the *ifnN* value to 0, else set *ifnN* to a valid function table number. When table translation is enabled (i.e. setting *ifnN* value to a non-zero number referring to an already allocated function table), *initN* value should be set equal to *iminN* or *imaxN* value, else the initial output value will not be the same as specified in *initN* argument.

*slider8* allows a bank of 8 different MIDI control message numbers.

As the input and output arguments are many, you can split the line using `\` (backslash) character (new in 3.47 version) to improve the readability. Using these opcodes is considerably more efficient than using the separate ones (*ctrl7* and *tonek*) when more controllers are required.

In the i-rate version of *slider8*, there is not an initial value input argument, because the output is gotten directly from current status of internal controller array of Csound.

## See Also

*s16b14, s32b14, slider16, slider16f, slider32, slider32f, slider64, slider64f, slider8f*

## Credits

Author: Gabriel Maldonado  
Italy  
December 1998

New in Csound version 3.50

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# slider8f

slider8f — Creates a bank of 8 different MIDI control message numbers, filtered before output.

## Description

Creates a bank of 8 different MIDI control message numbers, filtered before output.

## Syntax

```
k1,...,k8 slider8f ichan, ictlnum1, imin1, imax1, init1, ifn1, icutoff1, \  
..., ictlnum8, imin8, imax8, init8, ifn8, icutoff8
```

## Initialization

*ichan* -- MIDI channel (1-16)

*ictlnum1* ... *ictlnum8* -- MIDI control number (0-127)

*imin1* ... *imin8* -- minimum values for each controller

*imax1* ... *imax8* -- maximum values for each controller

*init1* ... *init8* -- initial value for each controller

*ifn1* ... *ifn8* -- function table for conversion for each controller

*icutoff1* ... *icutoff8* -- low-pass filter cutoff frequency for each controller

## Performance

*k1* ... *k8* -- output values

*slider8f* is a bank of MIDI controllers, useful when using MIDI mixer such as Kawai MM-16 or others for changing whatever sound parameter in real-time. The raw MIDI control messages at the input port are converted to agree with *iminN* and *imaxN*, and an initial value can be set. Also, an optional non-interpolated function table with a custom translation curve is allowed, useful for enabling exponential response curves.

When no function table translation is required, set the *ifnN* value to 0, else set *ifnN* to a valid function table number. When table translation is enabled (i.e. setting *ifnN* value to a non-zero number referring to an already allocated function table), *initN* value should be set equal to *iminN* or *imaxN* value, else the initial output value will not be the same as specified in *initN* argument.

*slider8f* allows a bank of 8 different MIDI control message numbers. It filters the signal before output. This eliminates discontinuities due to the low resolution of the MIDI (7 bit). The cutoff frequency can be set separately for each controller (suggested range: .1 to 5 Hz).

As the input and output arguments are many, you can split the line using `\` (backslash) character (new in 3.47 version) to improve the readability. Using these opcodes is considerably more efficient than using the separate ones (*ctrl7* and *tonek*) when more controllers are required.



## Warning

*slider8f* opcodes do not output the required initial value immediately, but only after some k-cycles because the filter slightly delays the output.

## See Also

*s16b14*, *s32b14*, *slider16*, *slider16f*, *slider32*, *slider32f*, *slider64*, *slider64f*, *slider8*

## Credits

Author: Gabriel Maldonado  
Italy  
December 1998

New in Csound version 3.50

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# slider8table

slider8table — Stores a bank of 8 different MIDI control messages to a table.

## Description

Stores a bank of 8 different MIDI control messages to a table.

## Syntax

```
kflag slider8table ichan, ioutTable, ioffset, ictlnum1, imin1, imax1, \  
    init1, ifn1, ..., ictlnum8, imin8, imax8, init8, ifn8
```

## Initialization

*i1 ... i8* -- output values

*ichan* -- MIDI channel (1-16)

*ioutTable* -- number of the table that will contain the output

*ioffset* -- output table offset. A zero means that the output of the first slider will affect the first table element. A 10 means that the output of the first slider will affect the 11th table element.

*ictlnum1 ... ictlnum8* -- MIDI control number (0-127)

*imin1 ... imin8* -- minimum values for each controller

*imax1 ... imax8* -- maximum values for each controller

*init1 ... init8* -- initial value for each controller

*ifn1 ... ifn8* -- function table for conversion for each controller

## Performance

*kflag* -- a flag that informs if any control-change message in the bank has been received. In this case *kflag* is set to 1. Otherwise is set to zero.

*slider8table* handles a bank of MIDI controllers, useful when using MIDI mixer such as Kawai MM-16 or others for changing whatever sound parameter in real-time. The raw MIDI control messages at the input port are converted to agree with *iminN* and *imaxN*, and an initial value can be set. Also, an optional non-interpolated function table with a custom translation curve is allowed, useful for enabling exponential response curves.

When no function table translation is required, set the *ifnN* value to 0, else set *ifnN* to a valid function table number. When table translation is enabled (i.e. setting *ifnN* value to a non-zero number referring to an already allocated function table), *initN* value should be set equal to *iminN* or *imaxN* value, else the initial output value will not be the same as specified in *initN* argument.

*slider8table* allows a bank of 8 different MIDI control message numbers.

As the input and output arguments are many, you can split the line using `\` (backslash) character (new in 3.47 version) to improve the readability. Using these opcodes is considerably more efficient than using the separate ones (*ctrl7* and *tonek*) when more controllers are required.

*slider8table* is very similar to *slider8* and *sliderN* family of opcodes (see their manual for more information). The actual difference is that the output is not stored to k-rate variables, but to a table, denoted by the *ioutTable* argument. It is possible to define a starting index in order to use the same table for more than one slider bank (or other purposes).

It is possible to use this opcode together with *FLslidBnk2Setk* and *FLslidBnk2*, so you can synchronize the position of the MIDI values to the position of the FLTK valuator widgets of *FLslidBnk2*. Notice that you have to specify the same min/max values as well the linear/exponential responses in both *sliderNtable(f)* and *FLslidBnk2*. The exception is when using table-indexed response instead of a lin/exp response. In this case, in order to achieve a useful result, the table-indexed response and actual min/max values must be set only in *FLslidBnk2*, whereas, in *sliderNtable(f)*, you have to set a linear response and a minimum of zero and a maximum of one in all sliders.

## See Also

*slider16table*, *slider16tablef*, *slider32table*, *slider32tablef*, *slider64table*, *slider64tablef*, *slider8tabletablef*

## Credits

Author: Gabriel Maldonado

New in Csound version 5.06

# slider8tablef

slider8tablef — Stores a bank of 8 different MIDI control messages to a table, filtered before output.

## Description

Stores a bank of 8 different MIDI control messages to a table, filtered before output.

## Syntax

```
kflag slider8tablef ichan, ioutTable, ioffset, ictlnum1, imin1, imax1, \  
init1, ifn1, icutoff1, ... , ictlnum8, imin8, imax8, init8, ifn8, icutoff8
```

## Initialization

*ichan* -- MIDI channel (1-16)

*ioutTable* -- number of the table that will contain the output

*ioffset* -- output table offset. A zero means that the output of the first slider will affect the first table element. A 10 means that the output of the first slider will affect the 11th table element.

*ictlnum1* ... *ictlnum8* -- MIDI control number (0-127)

*imin1* ... *imin8* -- minimum values for each controller

*imax1* ... *imax8* -- maximum values for each controller

*init1* ... *init8* -- initial value for each controller

*ifn1* ... *ifn8* -- function table for conversion for each controller

*icutoff1* ... *icutoff8* -- low-pass filter cutoff frequency for each controller

## Performance

*kflag* -- a flag that informs if any control-change message in the bank has been received. In this case *kflag* is set to 1. Otherwise is set to zero.

*slider8tablef* is a bank of MIDI controllers, useful when using MIDI mixer such as Kawai MM-16 or others for changing whatever sound parameter in real-time. The raw MIDI control messages at the input port are converted to agree with *iminN* and *imaxN*, and an initial value can be set. Also, an optional non-interpolated function table with a custom translation curve is allowed, useful for enabling exponential response curves.

When no function table translation is required, set the *ifnN* value to 0, else set *ifnN* to a valid function table number. When table translation is enabled (i.e. setting *ifnN* value to a non-zero number referring to an already allocated function table), *initN* value should be set equal to *iminN* or *imaxN* value, else the initial output value will not be the same as specified in *initN* argument.

*slider8tablef* allows a bank of 8 different MIDI control message numbers. It filters the signal before output. This eliminates discontinuities due to the low resolution of the MIDI (7 bit). The cutoff frequency can be set separately for each controller (suggested range: .1 to 5 Hz).

As the input and output arguments are many, you can split the line using `\` (backslash) character (new in 3.47 version) to improve the readability. Using these opcodes is considerably more efficient than using the separate ones (*ctrl7* and *tonek*) when more controllers are required.

*slider8tablef* is very similar to *slider8f* and *sliderNf* family of opcodes (see their manual for more information). The actual difference is that the output is not stored to k-rate variables, but to a table, denoted by the *ioutTable* argument. It is possible to define a starting index in order to use the same table for more than one slider bank (or other purposes).

It is possible to use this opcode together with *FLslidBnk2Setk* and *FLslidBnk2*, so you can synchronize the position of the MIDI values to the position of the FLTK valuator widgets of *FLslidBnk2*. Notice that you have to specify the same min/max values as well the linear/exponential responses in both *sliderNtable(f)* and *FLslidBnk2*. The exception is when using table-indexed response instead of a lin/exp response. In this case, in order to achieve a useful result, the table-indexed response and actual min/max values must be set only in *FLslidBnk2*, whereas, in *sliderNtable(f)*, you have to set a linear response and a minimum of zero and a maximum of one in all sliders.



### Warning

*slider8tablef* opcodes do not output the required initial value immediately, but only after some k-cycles because the filter slightly delays the output.

## See Also

*slider16table*, *slider16tablef*, *slider32table*, *slider32tablef*, *slider64table*, *slider64tablef*, *slider8table*

## Credits

Author: Gabriel Maldonado

New in Csound version 5.06

# sliderKawai

`sliderKawai` — Creates a bank of 16 different MIDI control message numbers from a KAWAI MM-16 midi mixer.

## Description

Creates a bank of 16 different MIDI control message numbers from a KAWAI MM-16 midi mixer.

## Syntax

```
k1, k2, ..., k16 sliderKawai imin1, imax1, init1, ifn1, \  
imin2, imax2, init2, ifn2, ..., imin16, imax16, init16, ifn16
```

## Initialization

*imin1* ... *imin16* -- minimum values for each controller

*imax1* ... *imax16* -- maximum values for each controller

*init1* ... *init16* -- initial value for each controller

*ifn1* ... *ifn16* -- function table for conversion for each controller

## Performance

*k1* ... *k16* -- output values

The opcode *sliderKawai* is equivalent to *slider16*, but it has the controller and channel numbers (*ichan* and *ictlnum*) hard-coded to make for quick compatibility with the KAWAI MM-16 midi mixer. This device doesn't allow changing the midi message associated to each slider. It can only output on control 7 for each fader on a separate midi channel. This opcode is a quick way of assigning the mixer's 16 faders to k-rate variables in csound.

## See Also

*slider16*, *slider16f*, *slider32*, *slider32f*, *slider64*, *slider64f*, *slider8*, *slider8f*

## Credits

Author: Gabriel Maldonado

New in Csound version 5.06

# sndloop

sndloop — A sound looper with pitch control.

## Description

This opcode records input audio and plays it back in a loop with user-defined duration and crossfade time. It also allows the pitch of the loop to be controlled, including reversed playback.

## Syntax

```
asig, krec sndloop ain, kpitch, ktrig, idur, ifad
```

## Initialization

*idur* -- loop duration in seconds

*ifad* -- crossfade duration in seconds

## Performance

*asig* -- output sig

*krec* -- 'rec on' signal, 1 when recording, 0 otherwise

*kpitch* -- pitch control (transposition ratio); negative values play the loop back in reverse

*ktrig* -- trigger signal: when 0, processing is bypassed. When switched on ( $ktrig \geq 1$ ), the opcode starts recording until the loop memory is full. It then plays the looped sound until it is switched off again ( $ktrig = 0$ ). Another recording can start again with  $ktrig \geq 1$ .

## Examples

Here is an example of the sndloop opcode. It uses the file *sndloop.csd* [examples/sndloop.csd].

### Example 899. Example of the sndloop opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o sndloop.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 1
```

```
nchnls = 2

instr 1

itrig = p4
asig disk2 "beats.wav", 1, 0, 1 ;get the signal in, loop it
ktrig line 0, itrig, 1 ;when to trigger signal = p4
kpitch line 1.2, p3, .5 ;vary pitch of recorded signal
aout,krec sndloop asig, kpitch, ktrig, .4, 0.05 ;rec starts at p4 sec, for .4 secs and 0.05 crossfade
printk2 krec ; prints the recording signal
outs aout, aout

endin

</CsInstruments>
<CsScore>

i1 0 5 .5 ;trigger in seconds (=p4)
i1 + 5 .8
i1 + 10 1.2

e
</CsScore>
</CsoundSynthesizer>
```

The example above shows the basic operation of `sndloop`. Pitch can be controlled at the k-rate, recording is started as soon as the trigger value is  $\geq 1$ . Recording can be restarted by making the trigger 0 and then 1 again.

## Credits

Author: Victor Lazzarini

April 2005

New in Version 5.00

# sndwarp

sndwarp — Reads a mono sound sample from a table and applies time-stretching and/or pitch modification.

## Description

*sndwarp* reads sound samples from a table and applies time-stretching and/or pitch modification. Time and frequency modification are independent from one another. For example, a sound can be stretched in time while raising the pitch!

The window size and overlap arguments are important to the result and should be experimented with. In general they should be as small as possible. For example, start with *iwsiz*e=*sr*/10 and *ioverlap*=15. Try *irandw*=*iwsiz*e\*0.2. If you can get away with less overlaps, the program will be faster. But too few may cause an audible flutter in the amplitude. The algorithm reacts differently depending upon the input sound and there are no fixed rules for the best use in all circumstances. But with proper tuning, excellent results can be achieved.

## Syntax

```
ares [, ac] sndwarp xamp, xtimewarp, xresample, ifn1, ibeg, iwsiz, \  
irandw, ioverlap, ifn2, itimemode
```

## Initialization

*ifn1* -- the number of the table holding the sound samples which will be subjected to the *sndwarp* processing. *GEN01* is the appropriate function generator to use to store the sound samples from a pre-existing soundfile.

*ibeg* -- the time in seconds to begin reading in the table (or soundfile). When *itimemode* is non-zero, the value of *xtimewarp* is offset by *ibeg*.

*iwsiz*e -- the window size in samples used in the time scaling algorithm.

*irandw* -- the bandwidth of a random number generator. The random numbers will be added to *iwsiz*e.

*ioverlap* -- determines the density of overlapping windows.

*ifn2* -- a function used to shape the window. It is usually used to create a ramp of some kind from zero at the beginning and back down to zero at the end of each window. Try using a half sine (i.e.: f1 0 16384 9 .5 1 0) which works quite well. Other shapes can also be used.

## Performance

*ares* -- the single channel of output from the *sndwarp* unit generator. *sndwarp* assumes that the function table holding the sampled signal is a mono one. This simply means that *sndwarp* will index the table by single-sample frame increments. The user must be aware then that if a stereo signal is used with *sndwarp*, time and pitch will be altered accordingly.

*ac* (optional) -- a single-layer (no overlaps), unwindowed versions of the time and/or pitch altered signal. They are supplied in order to be able to balance the amplitude of the signal output, which typically contains many overlapping and windowed versions of the signal, with a clean version of the time-scaled and pitch-shifted signal. The *sndwarp* process can cause noticeable changes in amplitude, (up and down), due to a

time differential between the overlaps when time-shifting is being done. When used with a *balance* unit, *ac* can greatly enhance the quality of sound.

*xamp* -- the value by which to scale the amplitude (see note on the use of this when using *ac*).

*xtimewarp* -- determines how the input signal will be stretched or shrunk in time. There are two ways to use this argument depending upon the value given for *itimemode*. When the value of *itimemode* is 0, *xtimewarp* will scale the time of the sound. For example, a value of 2 will stretch the sound by 2 times. When *itimemode* is any non-zero value then *xtimewarp* is used as a time pointer in a similar way in which the time pointer works in *lpread* and *pvoc*. An example below illustrates this. In both cases, the pitch will *not* be altered by this process. Pitch shifting is done independently using *xresample*.

*xresample* -- the factor by which to change the pitch of the sound. For example, a value of 2 will produce a sound one octave higher than the original. The timing of the sound, however, will *not* be altered.

## Examples

### Examples

Here is an example of the *sndwarp* opcode. It uses the file *sndwarp.csd* [examples/sndwarp.csd].

#### Example 900. Example of the *sndwarp* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o sndwarp.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
Odbfs = 1
nchnls = 2

instr 1

ktimewarp line 0, p3, 2.7 ;length of "fox.wav"
kresample init 1 ;do not change pitch
ibeg = 0 ;start at beginning
iysize = 4410 ;window size in samples with
irandw = 882 ;bandwidth of a random number generator
itimemode = 1 ;ktimewarp is "time" pointer
ioverlap = p4

asig sndwarp .5, ktimewarp, kresample, 1, ibeg, iysize, irandw, ioverlap, 2, itimemode
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 131072 1 "fox.wav" 0 0 0 ; audio file
f 2 0 1024 9 0.5 1 0 ; half of a sine wave
```

```

i 1 0 7 2 ;different overlaps
i 1 + 7 5
i 1 + 7 15
e

```

```

</CsScore>
</CsoundSynthesizer>

```

The below example shows a slowing down or stretching of the sound stored in the stored table (*ifn1*). Over the duration of the note, the stretching will grow from no change from the original to a sound which is ten times “slower” than the original. At the same time the overall pitch will move upward over the duration by an octave.

```

iwindfun = 1
isampfun = 2
ibeg = 0
iwindsize = 2000
iwindrand = 400
ioverlap = 10
awarp line 1, p3, 1
aresamp line 1, p3, 2
kenv line 1, p3, .1
asig sndwarp kenv, awarp, aresamp, isampfun, ibeg, iwindsize, iwindrand, ioverlap, iwindfun, 0

```

Now, here's an example using *xtimewarp* as a time pointer and using stereo:

```

itimemode = 1
atime line 0, p3, 10
ar1, ar2 sndwarpst kenv, atime, aresamp, sampfun, ibeg, iwindsize, iwindrand, ioverlap, \
          iwindfun, itimemode

```

In the above, *atime* advances the time pointer used in the *sndwarpst* from 0 to 10 over the duration of the note. If *p3* is 20 then the sound will be two times slower than the original. Of course you can use a more complex function than just a single straight line to control the time factor.

Now the same as above but using the *balance* function with the optional outputs:

```

asig,acmp sndwarp 1, awarp, aresamp, isampfun, ibeg, iwindsize, iwindrand, ioverlap, iwindfun, itimemode
abal balance asig, acmp

asig1,asig2,acmp1,acmp2 sndwarpst 1, atime, aresamp, sampfun, ibeg, iwindsize, iwindrand, ioverlap, \
                          iwindfun, itimemode
abal1 balance asig1, acmp1
abal2 balance asig2, acmp2

```

In the above two examples notice the use of the *balance* unit. The output of *balance* can then be scaled, enveloped, sent to an *out* or *outs*, and so on. Notice that the amplitude arguments to *sndwarp* and *sndwarpst* are “1” in these examples. By scaling the signal after the *sndwarp* process, *abal*, *abal1*, and *abal2* should contain signals that have nearly the same amplitude as the original input signal to the *sndwarp* process. This makes it much easier to predict the levels and avoid samples out of range or sample values that are too small.



## More Advice

Only use the stereo version when you really need to be processing a stereo file. It is somewhat slower than the mono version and if you use the *balance* function it is slower again. There is nothing wrong with using a mono *sndwarp* in a stereo orchestra and sending the result to one or both channels of the stereo output!

## See Also

*sndwarpst*

## Credits

Author: Richard Karpen  
Seattle, WA USA  
1997

# sndwarpst

`sndwarpst` — Reads a stereo sound sample from a table and applies time-stretching and/or pitch modification.

## Description

`sndwarpst` reads stereo sound samples from a table and applies time-stretching and/or pitch modification. Time and frequency modification are independent from one another. For example, a sound can be stretched in time while raising the pitch!

The window size and overlap arguments are important to the result and should be experimented with. In general they should be as small as possible. For example, start with `iwsize=sr/10` and `ioverlap=15`. Try `irandw=iwsize*.2`. If you can get away with less overlaps, the program will be faster. But too few may cause an audible flutter in the amplitude. The algorithm reacts differently depending upon the input sound and there are no fixed rules for the best use in all circumstances. But with proper tuning, excellent results can be achieved.

## Syntax

```
ar1, ar2 [,ac1] [, ac2] sndwarpst xamp, xtimewarp, xresample, ifn1, \  
ibeg, iwsize, irandw, ioverlap, ifn2, itimemode
```

## Initialization

`ifn1` -- the number of the table holding the sound samples which will be subjected to the `sndwarpst` processing. `GEN01` is the appropriate function generator to use to store the sound samples from a pre-existing soundfile.

`ibeg` -- the time in seconds to begin reading in the table (or soundfile). When `itimemode` is non-zero, the value of `xtimewarp` is offset by `ibeg`.

`iwsize` -- the window size in samples used in the time scaling algorithm.

`irandw` -- the bandwidth of a random number generator. The random numbers will be added to `iwsize`.

`ioverlap` -- determines the density of overlapping windows.

`ifn2` -- a function used to shape the window. It is usually used to create a ramp of some kind from zero at the beginning and back down to zero at the end of each window. Try using a half a sine (i.e.: `f1 0 16384 9 .5 1 0`) which works quite well. Other shapes can also be used.

## Performance

`ar1, ar2` -- `ar1` and `ar2` are the stereo (left and right) outputs from `sndwarpst`. `sndwarpst` assumes that the function table holding the sampled signal is a stereo one. `sndwarpst` will index the table by a two-sample frame increment. The user must be aware then that if a mono signal is used with `sndwarpst`, time and pitch will be altered accordingly.

`ac1, ac2` -- `ac1` and `ac2` are single-layer (no overlaps), unwindowed versions of the time and/or pitch altered signal. They are supplied in order to be able to balance the amplitude of the signal output, which typically contains many overlapping and windowed versions of the signal, with a clean version of the time-

scaled and pitch-shifted signal. The *sndwarpst* process can cause noticeable changes in amplitude, (up and down), due to a time differential between the overlaps when time-shifting is being done. When used with a *balance* unit, *ac1* and *ac2* can greatly enhance the quality of sound. They are optional, but note that they must both be present in the syntax (use both or neither). An example of how to use this is given below.

*xamp* -- the value by which to scale the amplitude (see note on the use of this when using *ac1* and *ac2*).

*xtimewarp* -- determines how the input signal will be stretched or shrunk in time. There are two ways to use this argument depending upon the value given for *itimemode*. When the value of *itimemode* is 0, *xtimewarp* will scale the time of the sound. For example, a value of 2 will stretch the sound by 2 times. When *itimemode* is any non-zero value then *xtimewarp* is used as a time pointer in a similar way in which the time pointer works in *lpread* and *pvoc*. An example below illustrates this. In both cases, the pitch will *not* be altered by this process. Pitch shifting is done independently using *xresample*.

*xresample* -- the factor by which to change the pitch of the sound. For example, a value of 2 will produce a sound one octave higher than the original. The timing of the sound, however, will *not* be altered.

## Example

Here is an example of the *sndwarpst* opcode. It uses the file *sndwarpst.csd* [examples/sndwarpst.csd].

### Example 901. Example of the *sndwarpst* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o sndwarpst.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ktimewarp line 0, p3, 1 ;length of stereo file "kickroll.wav"
kresample init 1 ;playback at the normal speed
ibeg = 0
iwsiz = 4410
irandw = 441
ioverlap = p4
itimemode = 1 ; Use the ktimewarp parameter as a "time" pointer

aL, aR sndwarpst .3, ktimewarp, kresample, 1, ibeg, iwsiz, irandw, ioverlap, 2, itimemode
aL dcblock aL ;get rid of DC offsets for left channel &
aR dcblock aR ;right channel
outs aL, aR

endin
</CsInstruments>
<CsScore>
f 1 0 65536 1 "kickroll.wav" 0 0 0
```

```
f 2 0 16384 9 0.5 1 0 ;half of a sine wave

i 1 0 7 2 ;different overlaps
i 1 + 7 5
i 1 + 7 15
e
</CsScore>
</CsoundSynthesizer>
```

## Other examples

The below example shows a slowing down or stretching of the sound stored in the stored table (*ifn1*). Over the duration of the note, the stretching will grow from no change from the original to a sound which is ten times “slower” than the original. At the same time the overall pitch will move upward over the duration by an octave.

```
iwindfun = 1
isampfun = 2
ibeg = 0
iwindsize = 2000
iwindrand = 400
ioverlap = 10
awarp line 1, p3, 1
aresamp line 1, p3, 2
kenv line 1, p3, .1
asig sndwarp kenv, awarp, aresamp, isampfun, ibeg, iwindsize, iwindrand, ioverlap, iwindfun, 0
```

Now, here's an example using *xtimewarp* as a time pointer and using stereo:

```
itimemode = 1
atime line 0, p3, 10
ar1, ar2 sndwarpst kenv, atime, aresamp, sampfun, ibeg, iwindsize, iwindrand, ioverlap, \
          iwindfun, itimemode
```

In the above, *atime* advances the time pointer used in the *sndwarpst* from 0 to 10 over the duration of the note. If *p3* is 20 then the sound will be two times slower than the original. Of course you can use a more complex function than just a single straight line to control the time factor.

Now the same as above but using the *balance* function with the optional outputs:

```
asig,acmp sndwarp 1, awarp, aresamp, isampfun, ibeg, iwindsize, iwindrand, ioverlap, iwindfun, itimemode
abal balance asig, acmp

asig1,asig2,acmp1,acmp2 sndwarpst 1, atime, aresamp, sampfun, ibeg, iwindsize, iwindrand, ioverlap, \
          iwindfun, itimemode

aball1 balance asig1, acmp1
abal2 balance asig2, acmp2
```

In the above two examples notice the use of the *balance* unit. The output of *balance* can then be scaled, enveloped, sent to an *out* or *outs*, and so on. Notice that the amplitude arguments to *sndwarp* and *sndwarpst* are “1” in these examples. By scaling the signal after the *sndwarp* process, *abal*, *abal1*, and *abal2* should contain signals that have nearly the same amplitude as the original input signal to the *sndwarp* process. This makes it much easier to predict the levels and avoid samples out of range or sample values that are too small.



### More Advice

Only use the stereo version when you really need to be processing a stereo file. It is somewhat slower than the mono version and if you use the *balance* function it is slower again. There

is nothing wrong with using a mono *sndwarp* in a stereo orchestra and sending the result to one or both channels of the stereo output!

## See Also

*sndwarp*

## Credits

Author: Richard Karpen  
Seattle, WA USA  
1997

# sockrecv

sockrecv — Receives data from other processes using the low-level UDP or TCP protocols

## Description

Receives directly using the UDP (*sockrecv* and *sockrecvs*) or TCP (*strecv*) protocol onto a network. The data is not subject to any encoding or special routing. The *sockrecvs* opcode receives a stereo signal interleaved.

## Syntax

```
asig sockrecv iport, ilength
ksig sockrecv iport, ilength
asigl, asigr sockrecvs iport, ilength
String sockrecv iport, ilength
asig strecv Sipaddr, iport
```

## Initialization

*Sipaddr* -- a string that is the IP address of the sender in standard 4-octet dotted form.

*iport* -- the number of the port that is used for the communication.

*ilength* -- the length of the individual packets in UDP transmission. This number must be sufficiently small to fit a single MTU, which is set to the save value of 1456. In UDP transmissions the sender and receiver needs agree on this value

## Performance

*asig*, *asigl*, *asigr* -- audio data to be received. *ksig* -- control data to be received. *String* -- string data to be received.

## Example

The example shows a mono signal being received on port 7777 using UDP.

```
sr = 44100
ksmps = 100
nchnls = 1

instr 1
al sockrecv 7777, 200
out al
endin
```

## Credits

Author: John ffitc

2006

# socksend

socksend — Sends data to other processes using the low-level UDP or TCP protocols

## Description

Transmits data directly using the UDP (*socksend* and *socksends*) or TCP (*stsend*) protocol onto a network. The data is not subject to any encoding or special routing. The *socksends* opcode send a stereo signal interleaved.

## Syntax

```
socksend asig, Sipaddr, iport, ilength
socksend ksig, Sipaddr, iport, ilength
socksends asigl, asigr, Sipaddr, iport,
           ilength
stsend asig, Sipaddr, iport
```

## Initialization

*Sipaddr* -- a string that is the IP address of the receiver in standard 4-octet dotted form.

*iport* -- the number of the port that is used for the communication.

*ilength* -- the length of the individual packets in UDP transmission. This number must be sufficiently small to fit a single MTU, which is set to the save value of 1456. In UDP transmissions the receiver needs to know this value

## Performance

*asig*, *ksig*, *asigl*, *asigr* -- data to be transmitted.

## Example

The example shows a simple sine wave being sent just once to a computer called "172.16.0.255", on port 7777 using UDP. Note that .255 is often used for broadcasting.

```
sr = 44100
ksmps = 100
nchnls = 1

instr 1
a1 oscil          20000,441,1
  socksend       a1, "172.16.0.255",7777, 200
endin
```

## Credits

Author: John ffitc

2006

# sorta

sorta — Sorts an array in ascending order.

## Description

Takes a numeric array (k or i-rate) and returns it sorted in ascending order.

## Syntax

```
k/i[]sorta k/i[] (k- or i-arrays )
```

## Examples

Here is an example of the sorta opcode. It uses the file *sorta.csd* [examples/sorta.csd].

### Example 902. Example of the sorta opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-n
</CsOptions>
<CsInstruments>

instr 1
  kArr[] fillarray 1,3,2,7,4
  kSorted[] sorta kArr
  kn = 0
  while kn < lenarray(kSorted) do
    printk2 kSorted[kn]
    kn += 1
  od
  turnoff
endin

</CsInstruments>
<CsScore>
i1 0 1
</CsScore>
</CsoundSynthesizer>
```

## See Also

*sortd*

## Credits

Author: Victor Lazzarini

New in version 6.09

# sortd

sortd — Sorts an array in descending order.

## Description

Takes a numeric array (k or i-rate) and returns it sorted in descending order.

## Syntax

```
k/i[]sortd k/i[] (k- or i-arrays )
```

## Examples

Here is an example of the sortd opcode. It uses the file *sortd.csd* [examples/sortd.csd].

### Example 903. Example of the sortd opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-n
</CsOptions>
<CsInstruments>

instr 1
  kArr[] fillarray 1,3,2,7,4
  kSorted[] sortd kArr
  kn = 0
  while kn < lenarray(kSorted) do
    printk2 kSorted[kn]
    kn += 1
  od
  turnoff
endin

</CsInstruments>
<CsScore>
i1 0 1
</CsScore>
</CsoundSynthesizer>
```

## See Also

*sorta*

## Credits

Author: Victor Lazzarini

New in version 6.09

# soundin

soundin — Reads audio data from an external device or stream.

## Description

Reads audio data from an external device or stream. Up to 24 channels may be read before v5.14, extended to 40 in later versions.

## Syntax

```
ar1[, ar2[, ar3[, ... a24]]] soundin ifilcod [, iskptim] [, iformat] \  
[, iskipinit] [, ibufsize]
```

## Initialization

*ifilcod* -- integer or character-string denoting the source soundfile name. An integer denotes the file soundin.filcod; a character-string (in double quotes, spaces permitted) gives the filename itself, optionally a full pathname. If not a full path, the named file is sought first in the current directory, then in that given by the environment variable *SSDIR* (if defined) then by *SFDIR*. See also *GENOI*.

*iskptim* (optional, default=0) -- time in seconds of input sound to be skipped. The default value is 0. In csound 5.00 and later, this may be negative to add a delay instead of skipping time.

*iformat* (optional, default=0) -- specifies the audio data file format:

- 1 = 8-bit signed char (high-order 8 bits of a 16-bit integer)
- 2 = 8-bit A-law bytes
- 3 = 8-bit U-law bytes
- 4 = 16-bit short integers
- 5 = 32-bit long integers
- 6 = 32-bit floats
- 7 = 8-bit unsigned int (not available in Csound versions older than 5.00)
- 8 = 24-bit int (not available in Csound versions older than 5.00)
- 9 = 64-bit doubles (not available in Csound versions older than 5.00)

*iskipinit* -- switches off all initialisation if non zero (default=0). This was introduced in 4\_23f13 and csound5.

*ibufsize* -- buffer size in mono samples (not sample frames). Not available in Csound versions older than 5.00. The default buffer size is 2048.

If *iformat* = 0 it is taken from the soundfile header, and if no header from the Csound *-o* command-line flag. The default value is 0.

## Performance

*soundin* is functionally an audio generator that derives its signal from a pre-existing file. The number of channels read in is controlled by the number of result cells, a1, a2, etc., which must match that of the input file. A *soundin* opcode opens this file whenever the host instrument is initialized, then closes it again each time the instrument is turned off.

There can be any number of *soundin* opcodes within a single instrument or orchestra. Two or more of them can read simultaneously from the same external file.



### Note to Windows users

Windows users typically use back-slashes, “\”, when specifying the paths of their files. As an example, a Windows user might use the path “c:\music\samples\loop001.wav”. This is problematic because back-slashes are normally used to specify special characters.

To correctly specify this path in Csound, one may alternately:

- Use forward slashes: c:/music/samples/loop001.wav
- Use back-slash special characters, “\\”: c:\\music\\samples\\loop001.wav

## Examples

Here is an example of the *soundin* opcode. It uses the file *soundin.csd* [examples/soundin.csd], *fox.wav* [examples/fox.wav] and *kickroll.wav* [examples/kickroll.wav].

### Example 904. Example of the *soundin* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
;-odac      ;;realtime audio out
-iadc      ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
;-o soundin.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; choose between mono or stereo file

ichn filenchnls p4 ;check number of channels
print  ichn

if ichn == 1 then
asig  soundin p4 ;mono signal
      outs  asig, asig
else  ;stereo signal
aL, aR soundin p4
      outs  aL, aR
```

```
endif

endin
</CsInstruments>
<CsScore>

i 1 0 3 "fox.wav" ;mono signal
i 1 5 2 "kickroll.wav" ;stereo signal

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*diskin, in, inh, ino, inq, ins*

## Credits

Authors: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

Warning to Windows users added by Kevin Conder, April 2002

# space

`space` — Distributes an input signal among 4 channels using cartesian coordinates.

## Description

`space` takes an input signal and distributes it among 4 channels using Cartesian xy coordinates to calculate the balance of the outputs. The xy coordinates can be defined in a separate text file and accessed through a Function statement in the score using `Gen28`, or they can be specified using the optional `kx`, `ky` arguments. The advantages to the former are:

1. A graphic user interface can be used to draw and edit the trajectory through the Cartesian plane
2. The file format is in the form `time1 X1 Y1 time2 X2 Y2 time3 X3 Y3` allowing the user to define a time-tagged trajectory

`space` then allows the user to specify a time pointer (much as is used for `pvoc`, `lpread` and some other units) to have detailed control over the final speed of movement.

## Syntax

```
a1, a2, a3, a4 space asig, ifn, ktime, kreverbsend, kx, ky
```

## Initialization

`ifn` -- number of the stored function created using `Gen28`. This function generator reads a text file which contains sets of three values representing the xy coordinates and a time-tag for when the signal should be placed at that location. The file should look like:

```
0   -1   1
1    1   1
2    4   4
2.1 -4  -4
3   10 -10
5  -40   0
```

If that file were named “move” then the `Gen28` call in the score would like:

```
f1 0 0 28 "move"
```

`Gen28` takes 0 as the size and automatically allocates memory. It creates values to 10 milliseconds of resolution. So in this case there will be 500 values created by interpolating X1 to X2 to X3 and so on, and Y1 to Y2 to Y3 and so on, over the appropriate number of values that are stored in the function table. In the above example, the sound will begin in the left front, over 1 second it will move to the right front, over another second it move further into the distance but still in the right front, then in just 1/10th of a second it moves to the left rear, a bit distant. Finally over the last .9 seconds the sound will move to the right rear, moderately distant, and it comes to rest between the two left channels (due west!), quite distant. Since the values in the table are accessed through the use of a time-pointer in the `space` unit, the actual

timing can be made to follow the file's timing exactly or it can be made to go faster or slower through the same trajectory. If you have access to the GUI that allows one to draw and edit the files, there is no need to create the text files manually. But as long as the file is ASCII and in the format shown above, it doesn't matter how it is made!



### Important

If *ifn* is 0, then *space* will take its values for the xy coordinates from *kx* and *ky*.

## Performance

The configuration of the xy coordinates in *space* places the signal in the following way:

- a1 is -1, 1
- a2 is 1, 1
- a3 is -1, -1
- a4 is 1, -1

This assumes a loudspeaker set up as a1 is left front, a2 is right front, a3 is left back, a4 is right back. Values greater than 1 will result in sounds being attenuated, as if in the distance. *space* considers the speakers to be at a distance of 1; smaller values of xy can be used, but *space* will not amplify the signal in this case. It will, however balance the signal so that it can sound as if it were within the 4 speaker space. x=0, y=1, will place the signal equally balanced between left and right front channels, x=y=0 will place the signal equally in all 4 channels, and so on. Although there must be 4 output signals from *space*, it can be used in a 2 channel orchestra. If the xy's are kept so that  $y \geq 1$ , it should work well to do panning and fixed localization in a stereo field.

*asig* -- input audio signal.

*ktime* -- index into the table containing the xy coordinates. If used like:

```
ktime      line 0, 5, 5
a1, a2, a3, a4 space asig, 1, ktime, ...
```

with the file "move" described above, the speed of the signal's movement will be exactly as described in that file. However:

```
ktime      line 0, 10, 5
```

the signal will move at half the speed specified. Or in the case of:

```
ktime      line 5, 15, 0
```

the signal will move in the reverse direction as specified and 3 times slower! Finally:

```
ktime      line 2, 10, 3
```

will cause the signal to move only from the place specified in line 3 of the text file to the place specified in line 5 of the text file, and it will take 10 seconds to do it.

*kreverb*send -- the percentage of the direct signal that will be factored along with the distance as derived from the xy coordinates to calculate signal amounts that can be sent to reverb units such as *reverb*, or *reverb2*.

*kx*, *ky* -- when *ifn* is 0, *space* and *spdist* will use these values as the xy coordinates to localize the signal.

## Examples

Here is an example of the space opcode. It uses the file *space\_quad.csd* [examples/space\_quad.csd].

### Example 905. Example of the space opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o space_quad.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
Odbfs = 1
nchnls = 4

ga1 init 0
ga2 init 0
ga3 init 0
ga4 init 0

instr 1 ;uses GEN28 file "move", as found in /manual/examples

kx   init 0
ky   init 0
ktime line 0, 5, 5 ;same time as in table 1 ("move")
asig diskin2 "beats.wav", 1, 0, 1 ;sound source is looped
a1, a2, a3, a4 space asig, 1, ktime, .1, kx, ky ;use table 1 = GEN28
ar1, ar2, ar3, ar4 spsend ;send to reverb

ga1 = ga1+ar1
ga2 = ga2+ar2
ga3 = ga3+ar3
ga4 = ga4+ar4
outq a1, a2, a3, a4

endin

instr 99 ; reverb instrument

a1 reverb2 ga1, 2.5, .5
a2 reverb2 ga2, 2.5, .5
a3 reverb2 ga3, 2.5, .5
```

```

a4 reverb2 ga4, 2.5, .5
   outq a1, a2, a3, a4

ga1=0
ga2=0
ga3=0
ga4=0

endin
</CsInstruments>
<CsScore>
f1 0 0 28 "move"

i1 0 5 ;same time as ktime
i 99 0 10 ;keep reverb active
e
</CsScore>
</CsoundSynthesizer>

```

In the above example, the signal, *asig*, is moved according to the data in Function #1 indexed by *ktime*. *space* sends the appropriate amount of the signal internally to *spsend*. The outputs of the *spsend* are added to global accumulators in a common Csound style and the global signals are used as inputs to the reverb units in a separate instrument.

*space* can be useful for quad and stereo panning as well as fixed placed of sounds anywhere between two loudspeakers. Below is an example of the fixed placement of sounds in a stereo field using *xy* values from the score instead of a function table. It uses the file *space\_stereo.csd* [examples/space\_stereo.csd].

### Example 906. Second example of the space opcode.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o space_stereo.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 2

ga1 init 0
ga2 init 0

instr 1

kx = p4
ky = p5
asig diskin2 "beats.wav", 1
a1, a2, a3, a4 space asig, 0, 0, .1, kx, ky ;take position values from p4, p5
ar1, ar2, ar3, ar4 spsend ;send to reverb

ga1 = ga1+ar1
ga2 = ga2+ar2
   outs a1, a2

endin

instr 99 ; reverb instrument

```

```
a1 reverb2 ga1, 2.5, .5
a2 reverb2 ga2, 2.5, .5
  outs a1, a2

ga1=0
ga2=0

endin

</CsInstruments>
<CsScore>
;place the sound in the left speaker and near
i1 0 1 -1 1
;place the sound in the right speaker and far
i1 1 1 45 45
;place the sound equally between left and right and in the middle ground distance
i1 2 1 0 12

i 99 0 7 ;keep reverb active
e
</CsScore>
</CsoundSynthesizer>
```

*spdist* demonstrates an example of a simple intuitive use of the distance values to simulate Doppler shift.

## See Also

*spdist*, *spsend*

## Credits

Author: Richard Karpen  
Seattle, WA USA  
1998

New in Csound version 3.48

# spat3d

spat3d — Positions the input sound in a 3D space and allows moving the sound at k-rate.

## Description

This opcode positions the input sound in a 3D space, with optional simulation of room acoustics, in various output formats. *spat3d* allows moving the sound at k-rate (this movement is interpolated internally to eliminate "zipper noise" if sr not equal to kr).

## Syntax

```
aW, aX, aY, aZ spat3d ain, kX, kY, kZ, idist, ift, imode, imdel, iovr [, istor]
```

## Initialization

*idist* -- For modes 0 to 3, *idist* is the unit circle distance in meters. For mode 4, *idist* is the distance between microphones.

The following formulas describe amplitude and delay as a function of sound source distance from microphone(s):

$$\text{amplitude} = 1 / (0.1 + \text{distance})$$

$$\text{delay} = \text{distance} / 340 \text{ (in seconds)}$$

Distance can be calculated as:

$$\text{distance} = \sqrt{iX^2 + iY^2 + iZ^2}$$

In Mode 4, distance can be calculated as:

$$\begin{aligned} \text{distance from left mic} &= \sqrt{(iX + idist/2)^2 + iY^2 + iZ^2} \\ \text{distance from right mic} &= \sqrt{(iX - idist/2)^2 + iY^2 + iZ^2} \end{aligned}$$

With *spat3d* the distance between the sound source and any microphone should be at least  $(340 * 18) / \text{sr}$  meters. Shorter distances will work, but may produce artifacts in some cases. There is no such limitation for *spat3di* and *spat3dt*.

Sudden changes or discontinuities in sound source location can result in pops or clicks. Very fast movement may also degrade quality.

*ift* -- Function table storing room parameters (for free field spatialization, set it to zero or negative). Table size is 54. The values in the table are:

Room Parameter	Purpose
0	Early reflection recursion depth (0 is the sound source, 1 is the first reflection etc.) for spat3d and spat3di. The number of echoes for four walls (front, back, right, left) is: $N = (2 * R + 2) * R$ . If all six walls are enabled: $N = (((4 * R + 6) * R + 8) * R) / 3$
1	Late reflection recursion depth (used by spat3dt only). spat3dt skips early reflections and renders echoes up to this level. If early reflection depth is negative, spat3d and spat3di will output zero, while spat3dt will start rendering from the sound source.
2	imdel for spat3d. Overrides opcode parameter if non-negative.
3	irlen for spat3dt. Overrides opcode parameter if non-negative.
4	idist value. Overrides opcode parameter if $\geq 0$ .
5	Random seed (0 - 65535) -1 seeds from current time.
6 - 53	wall parameters (w = 6: ceil, w = 14: floor, w = 22: front, w = 30: back, w = 38: right, w = 46: left)
w + 0	Enable reflections from this wall (0: no, 1: yes)
w + 1	Wall distance from listener (in meters)
w + 2	Randomization of wall distance (0 - 1) (in units of $1 / (\text{wall distance})$ )
w + 3	Reflection level (-1 - 1)
w + 4	Parametric equalizer frequency in Hz.
w + 5	Parametric equalizer level (1.0: no filtering)
w + 6	Parametric equalizer Q (0.7071: no resonance)
w + 7	Parametric equalizer mode (0: peak EQ, 1: low shelf, 2: high shelf)

*imode* -- Output mode

- 0: B format with W output only (mono)

aout = aW

- 1: B format with W and Y output (stereo)

aleft = aW + 0.7071\*aY

aright = aW - 0.7071\*aY

- 2: B format with W, X, and Y output (2D). This can be converted to UHJ:

aWre, aWim    hilbert aW

aXre, aXim    hilbert aX

aYre, aYim    hilbert aY

$$\begin{aligned} aWx_r &= 0.0928 * aX_{re} + 0.4699 * aW_{re} \\ aWx_iY_r &= 0.2550 * aX_{im} - 0.1710 * aW_{im} + 0.3277 * aY_{re} \\ aleft &= aWx_r + aWx_iY_r \\ aright &= aWx_r - aWx_iY_r \end{aligned}$$

- 3: B format with all outputs (3D)
- 4: Simulates a pair of microphones (stereo output)

$$\begin{aligned} aW & \text{ butterlp } aW, \text{ ifreq} \quad ; \text{ recommended values for ifreq} \\ aY & \text{ butterlp } aY, \text{ ifreq} \quad ; \text{ are around 1000 Hz} \\ aleft &= aW + aX \\ aright &= aY + aZ \end{aligned}$$

Mode 0 is the cheapest to calculate, while mode 4 is the most expensive.

In Mode 4, The optional lowpass filters can change the frequency response depending on direction. For example, if the sound source is located left to the listener then the high frequencies are attenuated in the right channel and slightly increased in the left. This effect can be disabled by not using filters. You can also experiment with other filters (tone etc.) for better effect.

Note that mode 4 is most useful for listening with headphones, and is also more expensive to calculate than the B-format (0 to 3) modes. The *idist* parameter in this case sets the distance between left and right microphone; for headphones, values between 0.2 - 0.25 are recommended, although higher settings up to 0.4 may be used for wide stereo effects.

More information about B format can be found here: [http://www.york.ac.uk/inst/mustech/3d\\_audio/am-bis2.htm](http://www.york.ac.uk/inst/mustech/3d_audio/am-bis2.htm)

*imdel* -- Maximum delay time for spat3d in seconds. This has to be longer than the delay time of the latest reflection (depends on room dimensions, sound source distance, and recursion depth; using this formula gives a safe (although somewhat overestimated) value:

$$\text{imdel} = (R + 1) * \text{sqrt}(W*W + H*H + D*D) / 340.0$$

where R is the recursion depth, W, H, and D are the width, height, and depth of the room, respectively).

*iovr* -- Oversample ratio for spat3d (1 to 8). Setting it higher improves quality at the expense of memory and CPU usage. The recommended value is 2.

*istor* (optional, default=0) -- Skip initialization if non-zero (default: 0).

## Performance

*aW, aX, aY, aZ* -- Output signals

	mode 0	mode 1	mode 2	mode 3	mode 4
aW	W out	W out	W out	W out	left chn / low freq.
aX	0	0	X out	X out	left chn / high freq.
aY	0	Y out	Y out	Y out	right chn / low freq.

	mode 0	mode 1	mode 2	mode 3	mode 4
aZ	0	0	0	Z out	right chn / high fr.

*ain* -- Input signal

*kX*, *kY*, *kZ* -- Sound source coordinates (in meters)

If you encounter very slow performance (up to 100 times slower), it may be caused by denormals (this is also true of many other IIR opcodes, including *butterlp*, *pareq*, *hilbert*, and many others). Underflows can be avoided by:

- Using the *denorm* opcode on *ain* before *spat3d*.
- mixing low level DC or noise to the input signal, e.g.

```
atmp rmd31 1/1e24, 0, 0
```

```
aW, aX, aY, aZ spa3di ain + atmp, ...
```

or

```
aW, aX, aY, aZ spa3di ain + 1/1e24, ...
```

- reducing *irlen* in the case of *spat3dt* (which does not have an input signal). A value of about 0.005 is suitable for most uses, although it also depends on EQ settings. If the equalizer is not used, “irlen” can be set to 0.

## Examples

Here is a example of the *spat3d* opcode that outputs a stereo file. It uses the file *spat3d\_stereo.csd* [examples/spat3d\_stereo.csd].

### Example 907. Stereo example of the *spat3d* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc    -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o spat3d_stereo.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

/* Written by Istvan Varga */
sr      = 48000
kr      = 1000
ksmps  = 48
nchnls = 2

/* room parameters */

idep    = 3      /* early reflection depth      */

itmp    ftgen  1, 0, 64, -2, \
          /* depth1, depth2, max delay, IR length, idist, seed */ \
```

## Orchestra Opcodes and Operators

```

idep, 48, -1, 0.01, 0.25, 123, \
1, 21.982, 0.05, 0.87, 4000.0, 0.6, 0.7, 2, /* ceil */ \
1, 1.753, 0.05, 0.87, 3500.0, 0.5, 0.7, 2, /* floor */ \
1, 15.220, 0.05, 0.87, 5000.0, 0.8, 0.7, 2, /* front */ \
1, 9.317, 0.05, 0.87, 5000.0, 0.8, 0.7, 2, /* back */ \
1, 17.545, 0.05, 0.87, 5000.0, 0.8, 0.7, 2, /* right */ \
1, 12.156, 0.05, 0.87, 5000.0, 0.8, 0.7, 2 /* left */

instr 1

/* some source signal */

a1      phasor 150           ; oscillator
a1      butterbp a1, 500, 200 ; filter
a1      = taninv(a1 * 100)
a2      phasor 3           ; envelope
a2      mirror 40*a2, -100, 5
a2      limit a2, 0, 1
a1      = a1 * a2 * 9000

kazim   line 0, 2.5, 360    ; move sound source around
kdist   line 1, 10, 4      ; distance

; convert polar coordinates
kX      = sin(kazim * 3.14159 / 180) * kdist
kY      = cos(kazim * 3.14159 / 180) * kdist
kZ      = 0

a1      = a1 + 0.000001 * 0.000001 ; avoid underflows

imode   = 1 ; change this to 3 for 8 spk in a cube,
; or 1 for simple stereo

aW, aX, aY, aZ spat3d a1, kX, kY, kZ, 1.0, 1, imode, 2, 2

aW      = aW * 1.4142

; stereo
;
aL      = aW + aY          /* left          */
aR      = aW - aY          /* right         */

; quad (square)
;
;aFL    = aW + aX + aY     /* front left   */
;aFR    = aW + aX - aY     /* front right  */
;aRL    = aW - aX + aY     /* rear left    */
;aRR    = aW - aX - aY     /* rear right   */

; eight channels (cube)
;
;aUFL   = aW + aX + aY + aZ /* upper front left */
;aUFR   = aW + aX - aY + aZ /* upper front right */
;aURL   = aW - aX + aY + aZ /* upper rear left  */
;aURR   = aW - aX - aY + aZ /* upper rear right  */
;aLFL   = aW + aX + aY - aZ /* lower front left  */
;aLFR   = aW + aX - aY - aZ /* lower front right */
;aLRL   = aW - aX + aY - aZ /* lower rear left   */
;aLRR   = aW - aX - aY - aZ /* lower rear right  */

outs aL, aR

endin

</CsInstruments>
<CsScore>

```

```

/* Written by Istvan Varga */
i 1 0 10
e

```

```

</CsScore>
</CsoundSynthesizer>

```

Here is an example of the spat3d opcode that outputs a UHJ file. It uses the file *spat3d\_UHJ.csd* [examples/spat3d\_UHJ.csd].

### Example 908. UHJ example of the spat3d opcode.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in   No messages
-odac         -iadc      -d          ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o spat3d_UHJ.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

/* Written by Istvan Varga */
sr = 48000
kr = 750
ksmps = 64
nchnls = 2

itmp   ftgen   1, 0, 64, -2,
/* depth1, depth2, max delay, IR length, idist, seed */ \
3, 48, -1, 0.01, 0.25, 123, \
1, 21.982, 0.05, 0.87, 4000.0, 0.6, 0.7, 2, /* ceil */ \
1, 1.753, 0.05, 0.87, 3500.0, 0.5, 0.7, 2, /* floor */ \
1, 15.220, 0.05, 0.87, 5000.0, 0.8, 0.7, 2, /* front */ \
1, 9.317, 0.05, 0.87, 5000.0, 0.8, 0.7, 2, /* back */ \
1, 17.545, 0.05, 0.87, 5000.0, 0.8, 0.7, 2, /* right */ \
1, 12.156, 0.05, 0.87, 5000.0, 0.8, 0.7, 2 /* left */

instr 1

p3 = p3 + 1.0

kazim line 0.0, 4.0, 360.0 ; azimuth
kelev line 40, p3 - 1.0, -20 ; elevation
kdist = 2.0 ; distance
; convert coordinates
kX = kdist * cos(kelev * 0.01745329) * sin(kazim * 0.01745329)
kY = kdist * cos(kelev * 0.01745329) * cos(kazim * 0.01745329)
kZ = kdist * sin(kelev * 0.01745329)

; source signal
a1 phasor 160.0
a2 delay1 a1
a1 = a1 - a2
kffrq1 port 200.0, 0.8, 12000.0
affrq upsamp kffrq1
affrq pareq affrq, 5.0, 0.0, 1.0, 2
kffrq downsamp affrq
aenv4 phasor 3.0
aenv4 limit 2.0 - aenv4 * 8.0, 0.0, 1.0
a1 butterbp a1 * aenv4, kffrq, 160.0
aenv linseg 1.0, p3 - 1.0, 1.0, 0.04, 0.0, 1.0, 0.0
a_ = 4000000 * a1 * aenv + 0.00000001

```

```

; spatialize
a_W, a_X, a_Y, a_Z spat3d a_, kX, kY, kZ, 1.0, 1, 2, 2.0, 2

; convert to UHJ format (stereo)
aWre, aWim hilbert a_W
aXre, aXim hilbert a_X
aYre, aYim hilbert a_Y

aWXre = 0.0928*aXre + 0.4699*aWre
aWXim = 0.2550*aXim - 0.1710*aWim

aL = aWXre + aWXim + 0.3277*aYre
aR = aWXre - aWXim - 0.3277*aYre

outs aL, aR

endin

</CsInstruments>
<CsScore>

/* Written by Istvan Varga */
t 0 60

i 1 0.0 8.0
e

</CsScore>
</CsoundSynthesizer>

```

Here is an example of the spat3d opcode that outputs a quadrophonic file. It uses the file *spat3d\_quad.csd* [examples/spat3d\_quad.csd].

### Example 909. Quadrophonic example of the spat3d opcode.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out Audio in No messages
-odac -iadc -d ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o spat3d_quad.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

/* Written by Istvan Varga */
sr = 48000
kr = 1000
ksmps = 48
nchnls = 4

/* room parameters */

idep = 3 /* early reflection depth */

itmp ftgen 1, 0, 64, -2,
/* depth1, depth2, max delay, IR length, idist, seed */ \
idep, 48, -1, 0.01, 0.25, 123, \
1, 21.982, 0.05, 0.87, 4000.0, 0.6, 0.7, 2, /* ceil */ \
1, 1.753, 0.05, 0.87, 3500.0, 0.5, 0.7, 2, /* floor */ \
1, 15.220, 0.05, 0.87, 5000.0, 0.8, 0.7, 2, /* front */ \
1, 9.317, 0.05, 0.87, 5000.0, 0.8, 0.7, 2, /* back */ \

```

## Orchestra Opcodes and Operators

---

```

1, 17.545, 0.05, 0.87, 5000.0, 0.8, 0.7, 2, /* right */ \
1, 12.156, 0.05, 0.87, 5000.0, 0.8, 0.7, 2 /* left */

instr 1

/* some source signal */

a1      phasor 150                ; oscillator
a1      butterbp a1, 500, 200    ; filter
a1      = taninv(a1 * 100)
a2      phasor 3                  ; envelope
a2      mirror 40*a2, -100, 5
a2      limit a2, 0, 1
a1      = a1 * a2 * 9000

kazim   line 0, 2.5, 360         ; move sound source around
kdist   line 1, 10, 4           ; distance

; convert polar coordinates
kX      = sin(kazim * 3.14159 / 180) * kdist
kY      = cos(kazim * 3.14159 / 180) * kdist
kZ      = 0

a1      = a1 + 0.000001 * 0.000001 ; avoid underflows

imode   = 2 ; change this to 3 for 8 spk in a cube,
; or 1 for simple stereo

aW, aX, aY, aZ spat3d a1, kX, kY, kZ, 1.0, 1, imode, 2, 2

aW      = aW * 1.4142

; stereo
;
;aL     = aW + aY                /* left          */
;aR     = aW - aY                /* right         */

; quad (square)
;
aFL     = aW + aX + aY           /* front left    */
aFR     = aW + aX - aY           /* front right   */
aRL     = aW - aX + aY           /* rear left     */
aRR     = aW - aX - aY           /* rear right    */

; eight channels (cube)
;
;aUFL   = aW + aX + aY + aZ     /* upper front left */
;aUFR   = aW + aX - aY + aZ     /* upper front right */
;aURL   = aW - aX + aY + aZ     /* upper rear left  */
;aURR   = aW - aX - aY + aZ     /* upper rear right */
;aLFL   = aW + aX + aY - aZ     /* lower front left */
;aLFR   = aW + aX - aY - aZ     /* lower front right */
;aLRL   = aW - aX + aY - aZ     /* lower rear left  */
;aLRR   = aW - aX - aY - aZ     /* lower rear right */

outq aFL, aFR, aRL, aRR

endin

</CsInstruments>
<CsScore>

/* Written by Istvan Varga */
t 0 60
i 1 0 10
e

```

```
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*spat3di, spat3dt*

## Credits

Author: Istvan Varga  
2001

New in version 4.12

Updated April 2002 by Istvan Varga

# spat3di

spat3di — Positions the input sound in a 3D space with the sound source position set at i-time.

## Description

This opcode positions the input sound in a 3D space, with optional simulation of room acoustics, in various output formats. With *spat3di*, sound source position is set at i-time.

## Syntax

```
aW, aX, aY, aZ spat3di ain, iX, iY, iZ, idist, ift, imode [, istor]
```

## Initialization

*iX* -- Sound source X coordinate in meters (positive: right, negative: left)

*iY* -- Sound source Y coordinate in meters (positive: front, negative: back)

*iZ* -- Sound source Z coordinate in meters (positive: up, negative: down)

*idist* -- For modes 0 to 3, *idist* is the unit circle distance in meters. For mode 4, *idist* is the distance between microphones.

The following formulas describe amplitude and delay as a function of sound source distance from microphone(s):

$$\text{amplitude} = 1 / (0.1 + \text{distance})$$
$$\text{delay} = \text{distance} / 340 \text{ (in seconds)}$$

Distance can be calculated as:

$$\text{distance} = \sqrt{iX^2 + iY^2 + iZ^2}$$

In Mode 4, distance can be calculated as:

$$\begin{aligned} \text{distance from left mic} &= \sqrt{(iX + idist/2)^2 + iY^2 + iZ^2} \\ \text{distance from right mic} &= \sqrt{(iX - idist/2)^2 + iY^2 + iZ^2} \end{aligned}$$

With *spat3d* the distance between the sound source and any microphone should be at least  $(340 * 18) / \text{sr}$  meters. Shorter distances will work, but may produce artifacts in some cases. There is no such limitation for *spat3di* and *spat3dt*.

Sudden changes or discontinuities in sound source location can result in pops or clicks. Very fast movement may also degrade quality.

*ift* -- Function table storing room parameters (for free field spatialization, set it to zero or negative). Table size is 54. The values in the table are:

Room Parameter	Purpose
0	Early reflection recursion depth (0 is the sound source, 1 is the first reflection etc.) for spat3d and spat3di. The number of echoes for four walls (front, back, right, left) is: $N = (2 * R + 2) * R$ . If all six walls are enabled: $N = (((4 * R + 6) * R + 8) * R) / 3$
1	Late reflection recursion depth (used by spat3dt only). spat3dt skips early reflections and renders echoes up to this level. If early reflection depth is negative, spat3d and spat3di will output zero, while spat3dt will start rendering from the sound source.
2	imdel for spat3d. Overrides opcode parameter if non-negative.
3	irlen for spat3dt. Overrides opcode parameter if non-negative.
4	idist value. Overrides opcode parameter if $\geq 0$ .
5	Random seed (0 - 65535) -1 seeds from current time.
6 - 53	wall parameters (w = 6: ceil, w = 14: floor, w = 22: front, w = 30: back, w = 38: right, w = 46: left)
w + 0	Enable reflections from this wall (0: no, 1: yes)
w + 1	Wall distance from listener (in meters)
w + 2	Randomization of wall distance (0 - 1) (in units of $1 / (\text{wall distance})$ )
w + 3	Reflection level (-1 - 1)
w + 4	Parametric equalizer frequency in Hz.
w + 5	Parametric equalizer level (1.0: no filtering)
w + 6	Parametric equalizer Q (0.7071: no resonance)
w + 7	Parametric equalizer mode (0: peak EQ, 1: low shelf, 2: high shelf)

*imode* -- Output mode

- 0: B format with W output only (mono)

$$aout = aW$$

- 1: B format with W and Y output (stereo)

$$\begin{aligned} aleft &= aW + 0.7071 * aY \\ aright &= aW - 0.7071 * aY \end{aligned}$$

- 2: B format with W, X, and Y output (2D). This can be converted to UHJ:

```

aWre, aWim  hilbert aW
aXre, aXim  hilbert aX
aYre, aYim  hilbert aY
aWXr  = 0.0928*aXre + 0.4699*aWre
aWXiYr = 0.2550*aXim - 0.1710*aWim + 0.3277*aYre
aleft  = aWXr + aWXiYr
aright = aWXr - aWXiYr

```

- 3: B format with all outputs (3D)
- 4: Simulates a pair of microphones (stereo output)

```

aW  butterlp aW, ifreq  ; recommended values for ifreq
aY  butterlp aY, ifreq  ; are around 1000 Hz
aleft = aW + aX
aright = aY + aZ

```

Mode 0 is the cheapest to calculate, while mode 4 is the most expensive.

In Mode 4, The optional lowpass filters can change the frequency response depending on direction. For example, if the sound source is located left to the listener then the high frequencies are attenuated in the right channel and slightly increased in the left. This effect can be disabled by not using filters. You can also experiment with other filters (tone etc.) for better effect.

Note that mode 4 is most useful for listening with headphones, and is also more expensive to calculate than the B-format (0 to 3) modes. The *idist* parameter in this case sets the distance between left and right microphone; for headphones, values between 0.2 - 0.25 are recommended, although higher settings up to 0.4 may be used for wide stereo effects.

More information about B format can be found here: [http://www.york.ac.uk/inst/mustech/3d\\_audio/am-bis2.htm](http://www.york.ac.uk/inst/mustech/3d_audio/am-bis2.htm)

*istor* (optional, default=0) -- Skip initialization if non-zero (default: 0).

## Performance

*ain* -- Input signal

*aW, aX, aY, aZ* -- Output signals

	mode 0	mode 1	mode 2	mode 3	mode 4
aW	W out	W out	W out	W out	left chn / low freq.
aX	0	0	X out	X out	left chn / high freq.
aY	0	Y out	Y out	Y out	right chn / low freq.
aZ	0	0	0	Z out	right chn / high fr.

If you encounter very slow performance (up to 100 times slower), it may be caused by denormals (this is also true of many other IIR opcodes, including *butterlp*, *pareq*, *hilbert*, and many others). Underflows can be avoided by:

- Using the *denorm* opcode on *ain* before *spat3di*.
- mixing low level DC or noise to the input signal, e.g.  

```
atmp rnd31 1/1e24, 0, 0
```

```
aW, aX, aY, aZ spat3di ain + atmp, ...
```

or  

```
aW, aX, aY, aZ spa3di ain + 1/1e24, ...
```
- reducing *irlen* in the case of *spat3dt* (which does not have an input signal). A value of about 0.005 is suitable for most uses, although it also depends on EQ settings. If the equalizer is not used, “*irlen*” can be set to 0.

## Examples

See the examples for *spat3d*.

## See Also

*spat3d*, *spat3dt*

## Credits

Author: Istvan Varga  
2001

New in version 4.12

Updated April 2002 by Istvan Varga

# spat3dt

`spat3dt` — Can be used to render an impulse response for a 3D space at i-time.

## Description

This opcode positions the input sound in a 3D space, with optional simulation of room acoustics, in various output formats. *spat3dt* can be used to render the impulse response at i-time, storing output in a function table, suitable for convolution.

## Syntax

```
spat3dt ioutft, iX, iY, iZ, idist, ift, imode, irlen [, iftnocl]
```

## Initialization

*ioutft* -- Output ftable number for `spat3dt`. W, X, Y, and Z outputs are written interleaved to this table. If the table is too short, output will be truncated.

*iX* -- Sound source X coordinate in meters (positive: right, negative: left)

*iY* -- Sound source Y coordinate in meters (positive: front, negative: back)

*iZ* -- Sound source Z coordinate in meters (positive: up, negative: down)

*idist* -- For modes 0 to 3, *idist* is the unit circle distance in meters. For mode 4, *idist* is the distance between microphones.

The following formulas describe amplitude and delay as a function of sound source distance from microphone(s):

amplitude =  $1 / (0.1 + \text{distance})$

delay = distance / 340 (in seconds)

Distance can be calculated as:

distance =  $\sqrt{iX^2 + iY^2 + iZ^2}$

In Mode 4, distance can be calculated as:

distance from left mic =  $\sqrt{(iX + idist/2)^2 + iY^2 + iZ^2}$

distance from right mic =  $\sqrt{(iX - idist/2)^2 + iY^2 + iZ^2}$

With *spat3d* the distance between the sound source and any microphone should be at least  $(340 * 18) / sr$  meters. Shorter distances will work, but may produce artifacts in some cases. There is no such limitation for *spat3di* and *spat3dt*.

Sudden changes or discontinuities in sound source location can result in pops or clicks. Very fast movement may also degrade quality.

*ift* -- Function table storing room parameters (for free field spatialization, set it to zero or negative). Table size is 54. The values in the table are:

Room Parameter	Purpose
0	Early reflection recursion depth (0 is the sound source, 1 is the first reflection etc.) for <i>spat3d</i> and <i>spat3di</i> . The number of echoes for four walls (front, back, right, left) is: $N = (2 * R + 2) * R$ . If all six walls are enabled: $N = (((4 * R + 6) * R + 8) * R) / 3$
1	Late reflection recursion depth (used by <i>spat3dt</i> only). <i>spat3dt</i> skips early reflections and renders echoes up to this level. If early reflection depth is negative, <i>spat3d</i> and <i>spat3di</i> will output zero, while <i>spat3dt</i> will start rendering from the sound source.
2	<i>imdel</i> for <i>spat3d</i> . Overrides opcode parameter if non-negative.
3	<i>irlen</i> for <i>spat3d</i> . Overrides opcode parameter if non-negative.
4	<i>idist</i> value. Overrides opcode parameter if $\geq 0$ .
5	Random seed (0 - 65535) -1 seeds from current time.
6 - 53	wall parameters (w = 6: ceil, w = 14: floor, w = 22: front, w = 30: back, w = 38: right, w = 46: left)
w + 0	Enable reflections from this wall (0: no, 1: yes)
w + 1	Wall distance from listener (in meters)
w + 2	Randomization of wall distance (0 - 1) (in units of $1 / (\text{wall distance})$ )
w + 3	Reflection level (-1 - 1)
w + 4	Parametric equalizer frequency in Hz.
w + 5	Parametric equalizer level (1.0: no filtering)
w + 6	Parametric equalizer Q (0.7071: no resonance)
w + 7	Parametric equalizer mode (0: peak EQ, 1: low shelf, 2: high shelf)

*imode* -- Output mode

- 0: B format with W output only (mono)

aout = aW

- 1: B format with W and Y output (stereo)

```

aleft = aW + 0.7071*aY
aright = aW - 0.7071*aY

```

- 2: B format with W, X, and Y output (2D). This can be converted to UHJ:

```

aWre, aWim    hilbert aW
aXre, aXim    hilbert aX
aYre, aYim    hilbert aY
aWXr = 0.0928*aXre + 0.4699*aWre
aWXiYr = 0.2550*aXim - 0.1710*aWim + 0.3277*aYre
aleft = aWXr + aWXiYr
aright = aWXr - aWXiYr

```

- 3: B format with all outputs (3D)
- 4: Simulates a pair of microphones (stereo output)

```

aW    butterlp aW, ifreq    ; recommended values for ifreq
aY    butterlp aY, ifreq    ; are around 1000 Hz
aleft = aW + aX
aright = aY + aZ

```

Mode 0 is the cheapest to calculate, while mode 4 is the most expensive.

In Mode 4, The optional lowpass filters can change the frequency response depending on direction. For example, if the sound source is located left to the listener then the high frequencies are attenuated in the right channel and slightly increased in the left. This effect can be disabled by not using filters. You can also experiment with other filters (tone etc.) for better effect.

Note that mode 4 is most useful for listening with headphones, and is also more expensive to calculate than the B-format (0 to 3) modes. The *idist* parameter in this case sets the distance between left and right microphone; for headphones, values between 0.2 - 0.25 are recommended, although higher settings up to 0.4 may be used for wide stereo effects.

More information about B format can be found here: [http://www.york.ac.uk/inst/mustech/3d\\_audio/ambis2.htm](http://www.york.ac.uk/inst/mustech/3d_audio/ambis2.htm)

*irlen* -- Impulse response length of echoes (in seconds). Depending on filter parameters, values around 0.005-0.01 are suitable for most uses (higher values result in more accurate output, but slower rendering)

*ifmocl* (optional, default=0) -- Do not clear output ftable (mix to existing data) if set to 1, clear table before writing if set to 0 (default: 0).

## Examples

Here is an example of the *spat3dt* opcode. It uses the file *spat3dt.csd* [examples/spat3dt.csd].

### Example 910. Example of the *spat3dt* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o spat3dt.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

garvb  init 0
gaW    init 0
gaX    init 0
gaY    init 0

itmp ftgen 1, 0, 64, -2, 2, 40, -1, -1, -1, 123,          \
      1, 13.000, 0.05, 0.85, 20000.0, 0.0, 0.50, 2,    \
      1,  2.000, 0.05, 0.85, 20000.0, 0.0, 0.25, 2,    \
      1, 16.000, 0.05, 0.85, 20000.0, 0.0, 0.35, 2,    \
      1,  9.000, 0.05, 0.85, 20000.0, 0.0, 0.35, 2,    \
      1, 12.000, 0.05, 0.85, 20000.0, 0.0, 0.35, 2,    \
      1,  8.000, 0.05, 0.85, 20000.0, 0.0, 0.35, 2

itmp ftgen 2, 0, 262144, -2, 0
      spat3dt 2, -0.2, 1, 0, 1, 1, 2, 0.005

itmp ftgen 3, 0, 262144, -52, 3, 2, 0, 4, 2, 1, 4, 2, 2, 4

instr 1

a1  vco2 1, 440, 10
kfrq port 100, 0.008, 20000
a1  butterlp a1, kfrq
a2  linseg 0, 0.003, 1, 0.01, 0.7, 0.005, 0, 1, 0
a1  = a1 * a2 * 2
      denorm a1
vincr garvb, a1
aw, ax, ay, az spat3di a1, p4, p5, p6, 1, 1, 2
      vincr gaW, aw
      vincr gaX, ax
      vincr gaY, ay

endin

instr 2

denorm garvb
; skip as many samples as possible without truncating the IR
arW, arX, arY ftconv garvb, 3, 2048, 2048, (65536 - 2048)
aW   = gaW + arW
aX   = gaX + arX
aY   = gaY + arY
garvb = 0
gaW   = 0
gaX   = 0
gaY   = 0

aWre, aWim hilbert aW
aXre, aXim hilbert aX
aYre, aYim hilbert aY

```

```
aWXr    = 0.0928*aXre + 0.4699*aWre
aWXiYr  = 0.2550*aXim - 0.1710*aWim + 0.3277*aYre
aL      = aWXr + aWXiYr
aR      = aWXr - aWXiYr
outs    aL, aR
```

```
endin
```

```
</CsInstruments>
<CsScore>
```

```
i 1 0 0.5 0.0 2.0 -0.8
i 1 1 0.5 1.4 1.4 -0.6
i 1 2 0.5 2.0 0.0 -0.4
i 1 3 0.5 1.4 -1.4 -0.2
i 1 4 0.5 0.0 -2.0 0.0
i 1 5 0.5 -1.4 -1.4 0.2
i 1 6 0.5 -2.0 0.0 0.4
i 1 7 0.5 -1.4 1.4 0.6
i 1 8 0.5 0.0 2.0 0.8
i 2 0 10
e
```

```
</CsScore>
</CsoundSynthesizer>
```

## See Also

*spat3d, spat3di*

## Credits

Author: Istvan Varga  
2001

New in version 4.12

Updated April 2002 by Istvan Varga

# spdist

spdist — Calculates distance values from xy coordinates.

## Description

*spdist* uses the same xy data as *space*, also either from a text file using *Gen28* or from x and y arguments given to the unit directly. The purpose of this unit is to make available the values for distance that are calculated from the xy coordinates.

In the case of *space*, the xy values are used to determine a distance which is used to attenuate the signal and prepare it for use in *spsend*. But it is also useful to have these values for distance available to scale the frequency of the signal before it is sent to the *space* unit.

## Syntax

```
k1 spdist ifn, ktime, kx, ky
```

## Initialization

*ifn* -- number of the stored function created using *Gen28*. This function generator reads a text file which contains sets of three values representing the xy coordinates and a time-tag for when the signal should be placed at that location. The file should look like:

```
0   -1   1
1    1   1
2    4   4
2.1 -4  -4
3    10 -10
5   -40   0
```

If that file were named "move" then the *Gen28* call in the score would like:

```
f1 0 0 28 "move"
```

*Gen28* takes 0 as the size and automatically allocates memory. It creates values to 10 milliseconds of resolution. So in this case there will be 500 values created by interpolating X1 to X2 to X3 and so on, and Y1 to Y2 to Y3 and so on, over the appropriate number of values that are stored in the function table. In the above example, the sound will begin in the left front, over 1 second it will move to the right front, over another second it move further into the distance but still in the right front, then in just 1/10th of a second it moves to the left rear, a bit distant. Finally over the last .9 seconds the sound will move to the right rear, moderately distant, and it comes to rest between the two left channels (due west!), quite distant. Since the values in the table are accessed through the use of a time-pointer in the *space* unit, the actual timing can be made to follow the file's timing exactly or it can be made to go faster or slower through the same trajectory. If you have access to the GUI that allows one to draw and edit the files, there is no need to create the text files manually. But as long as the file is ASCII and in the format shown above, it doesn't matter how it is made!

IMPORTANT: If *ifn* is 0 then *spdist* will take its values for the xy coordinates from *kx* and *ky*.

## Performance

The configuration of the xy coordinates in space places the signal in the following way:

- a1 is -1, 1
- a2 is 1, 1
- a3 is -1, -1
- a4 is 1, -1

This assumes a loudspeaker set up as a1 is left front, a2 is right front, a3 is left back, a4 is right back. Values greater than 1 will result in sounds being attenuated, as if in the distance. *space* considers the speakers to be at a distance of 1; smaller values of xy can be used, but *space* will not amplify the signal in this case. It will, however balance the signal so that it can sound as if it were within the 4 speaker space. x=0, y=1, will place the signal equally balanced between left and right front channels, x=y=0 will place the signal equally in all 4 channels, and so on. Although there must be 4 output signals from *space*, it can be used in a 2 channel orchestra. If the xy's are kept so that  $Y \geq 1$ , it should work well to do panning and fixed localization in a stereo field.

*ktime* -- index into the table containing the xy coordinates. If used like:

```
ktime      line 0, 5, 5
a1, a2, a3, a4 space asig, 1, ktime, ...
```

with the file "move" described above, the speed of the signal's movement will be exactly as described in that file. However:

```
ktime      line 0, 10, 5
```

the signal will move at half the speed specified. Or in the case of:

```
ktime      line 5, 15, 0
```

the signal will move in the reverse direction as specified and 3 times slower! Finally:

```
ktime      line 2, 10, 3
```

will cause the signal to move only from the place specified in line 3 of the text file to the place specified in line 5 of the text file, and it will take 10 seconds to do it.

*kx*, *ky* -- when *ifn* is 0, *space* and *spdist* will use these values as the XY coordinates to localize the signal.

## Examples

Here is an example of the *spdist* opcode. It uses the file *spdist.csd* [examples/spdist.csd].

**Example 911. Example of the spdist opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o spdist.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 4

ga1 init 0
ga2 init 0
ga3 init 0
ga4 init 0

instr 1 ;uses GEN28 file "move2", as found in /manual/examples

ifreq = 1
kx   init 0
ky   init 0
ktime line 0, 5.6, 5.6 ;same time as in table 1 ("move2")
kdist spdist 1, ktime, kx, ky
kfreq = (ifreq*340) / (340 + kdist) ;calculate doppler shift
printk2 kdist ;print distance values
asig disk2 "flute.aiff", kfreq, 0, 1 ;sound source is looped
a1, a2, a3, a4 space asig, 1, ktime, .1, kx, ky ;use table 1 = GEN28
ar1, ar2, ar3, ar4 spsend ;send to reverb

ga1 = ga1+ar1
ga2 = ga2+ar2
ga3 = ga3+ar3
ga4 = ga4+ar4
outq a1, a2, a3, a4

endin

instr 99 ; reverb instrument

a1 reverb2 ga1, 2.5, .5
a2 reverb2 ga2, 2.5, .5
a3 reverb2 ga3, 2.5, .5
a4 reverb2 ga4, 2.5, .5
outq a1, a2, a3, a4

ga1=0
ga2=0
ga3=0
ga4=0

endin
</CsInstruments>
<CsScore>
f1 0 0 28 "move2" ;from left front and left rear to the middle in front

i 1 0 5.6 ;same time as ktime
i 99 0 10 ;keep reverb active

```

```
e  
</CsScore>  
</CsoundSynthesizer>
```

The same function and time values are used for both *spdist* and *space*. This insures that the distance values used internally in the *space* unit will be the same as those returned by *spdist* to give the impression of a Doppler shift!

## See Also

*space*, *spsend*

## Credits

Author: Richard Karpen  
Seattle, WA USA  
1998

New in Csound version 3.48

# specaddm

specaddm — Perform a weighted add of two input spectra.

## Description

Perform a weighted add of two input spectra.

## Syntax

```
wsig specaddm wsig1, wsig2 [, imul2]
```

## Initialization

*imul2* (optional, default=0) -- if non-zero, scale the *wsig2* magnitudes before adding. The default value is 0.

## Performance

*wsig1* -- the first input spectra.

*wsig2* -- the second input spectra.

Do a weighted add of two input spectra. For each channel of the two input spectra, the two magnitudes are combined and written to the output according to:

$$\text{magout} = \text{mag1in} + \text{mag2in} * \text{imul2}$$

The operation is performed whenever the input *wsig1* is sensed to be new. This unit will (at Initialization) verify the consistency of the two spectra (equal size, equal period, equal mag types).

## See Also

*specdiff*, *specfilt*, *spechist*, *specscal*

# specdiff

`specdiff` — Finds the positive difference values between consecutive spectral frames.

## Description

Finds the positive difference values between consecutive spectral frames.

## Syntax

```
wsig specdiff wsignin
```

## Performance

*wsig* -- the output spectrum.

*wsignin* -- the input spectra.

Finds the positive difference values between consecutive spectral frames. At each new frame of *wsignin*, each magnitude value is compared with its predecessor, and the positive changes written to the output spectrum. This unit is useful as an energy onset detector.

## Examples

```
wsig2  specdiff  wsig1      ; sense onsets
wsig3  specfilt  wsig2, 2    ; absorb slowly
       specdisp  wsig2, 0.1  ; & display both spectra
       specdisp  wsig3, 0.1
```

## See Also

*specaddm*, *specfilt*, *spechist*, *specscal*

# specdisp

`specdisp` — Displays the magnitude values of the spectrum.

## Description

Displays the magnitude values of the spectrum.

## Syntax

```
specdisp wsig, iprd [, iwtflg]
```

## Initialization

`iprd` -- the period, in seconds, of each new display.

`iwtflg` (optional, default=0) -- wait flag. If non-zero, hold each display until released by the user. The default value is 0 (no wait).

## Performance

`wsig` -- the input spectrum.

Displays the magnitude values of spectrum `wsig` every `iprd` seconds (rounded to some integral number of `wsig`'s originating `iprd`).

## Examples

```
ksum    specsum  wsig, 1           ; sum the spec bins, and ksmooth
        if ksum < 2000 kgoto zero  ; if sufficient amplitude
koct    specptrk wsig             ; pitch-track the signal
        kgoto    contin
zero:
koct    =         0               ; else output zero
contin:
```

## See Also

`specsum`

# specfilt

specfilt — Filters each channel of an input spectrum.

## Description

Filters each channel of an input spectrum.

## Syntax

```
wsig specfilt wsignin, ifhtim
```

## Initialization

*ifhtim* -- half-time constant.

## Performance

*wsignin* -- the input spectrum.

Filters each channel of an input spectrum. At each new frame of *wsignin*, each magnitude value is injected into a 1st-order lowpass recursive filter, whose half-time constant has been initially set by sampling the ftable *ifhtim* across the (logarithmic) frequency space of the input spectrum. This unit effectively applies a *persistence* factor to the data occurring in each spectral channel, and is useful for simulating the *energy integration* that occurs during auditory perception. It may also be used as a time-attenuated running *histogram* of the spectral distribution.

## Examples

```
wsig2  specdiff  wsig1      ; sense onsets
wsig3  specfilt  wsig2, 2    ; absorb slowly
        specdisp  wsig2, 0.1 ; & display both spectra
        specdisp  wsig3, 0.1
```

## See Also

*specaddm*, *specdiff*, *spechist*, *specscal*

# spechist

spechist — Accumulates the values of successive spectral frames.

## Description

Accumulates the values of successive spectral frames.

## Syntax

```
wsig spechist wsignin
```

## Performance

*wsignin* -- the input spectra.

Accumulates the values of successive spectral frames. At each new frame of *wsignin*, the accumulations-to-date in each magnitude track are written to the output spectrum. This unit thus provides a running *histogram* of spectral distribution.

## Examples

```
wsig2  specdiff  wsig1      ; sense onsets
wsig3  specfilt  wsig2, 2    ; absorb slowly
        specdisp  wsig2, 0.1  ; & display both spectra
        specdisp  wsig3, 0.1
```

## See Also

*specadm*, *specdiff*, *specfilt*, *specscal*

# specptrk

specptrk — Estimates the pitch of the most prominent complex tone in the spectrum.

## Description

Estimate the pitch of the most prominent complex tone in the spectrum.

## Syntax

```
koct, kamp specptrk wsig, kvar, ilo, ihi, istr, idbthresh, inptls, \  
  irolloff [, ioddd] [, iconfs] [, interp] [, ifprd] [, iwftlg]
```

## Initialization

*ilo*, *ihi*, *istr* -- pitch range conditioners (low, high, and starting) expressed in decimal octave form.

*idbthresh* -- energy threshold (in decibels) for pitch tracking to occur. Once begun, tracking will be continuous until the energy falls below one half the threshold (6 dB down), whence the *koct* and *kamp* outputs will be zero until the full threshold is again surpassed. *idbthresh* is a guiding value. At initialization it is first converted to the *idbout* mode of the source spectrum (and the 6 dB down point becomes .5, .25, or 1/root 2 for modes 0, 2 and 3). The values are also further scaled to allow for the weighted partial summation used during correlation. The actual thresholding is done using the internal weighted and summed *kamp* value that is visible as the second output parameter.

*inptls*, *iroloff* -- number of harmonic partials used as a matching template in the spectrally-based pitch detection, and an amplitude rolloff for the set expressed as some fraction per octave (linear, so don't roll off to negative). Since the partials and rolloff fraction can affect the pitch following, some experimentation will be useful: try 4 or 5 partials with .6 rolloff as an initial setting; raise to 10 or 12 partials with rolloff .75 for complex timbres like the bassoon (weak fundamental). Computation time is dependent on the number of partials sought. The maximum number is 16.

*iodd* (optional) -- if non-zero, employ only odd partials in the above set (e.g. *inptls* of 4 would employ partials 1,3,5,7). This improves the tracking of some instruments like the clarinet. The default value is 0 (employ all partials).

*iconfs* (optional) -- number of confirmations required for the pitch tracker to jump an octave, pro-rated for fractions of an octave (i.e. the value 12 implies a semitone change needs 1 confirmation (two hits) at the *spectrum* generating *iprd*). This parameter limits spurious pitch analyses such as octave errors. A value of 0 means no confirmations required; the default value is 10.

*interp* (optional) -- if non-zero, interpolate each output signal (*koct*, *kamp*) between incoming *wsig* frames. The default value is 0 (repeat the signal values between frames).

*ifprd* (optional) -- if non-zero, display the internally computed spectrum of candidate fundamentals. The default value is 0 (no display).

*iwftg* (optional) -- wait flag. If non-zero, hold each display until released by the user. The default value is 0 (no wait).

## Performance

At note initialization this unit creates a template of *inptls* harmonically related partials (odd partials, if *iodd* non-zero) with amplitude rolloff to the fraction *iroloff* per octave. At each new frame of *wsig*, the

spectrum is cross-correlated with this template to provide an internal spectrum of candidate fundamentals (optionally displayed). A likely pitch/amp pair (*koct*, *kamp*, in decimal octave and summed *idbout* form) is then estimated. *koct* varies from the previous *koct* by no more than plus or minus *kvar* decimal octave units. It is also guaranteed to lie within the hard limit range *ilo* -- *ihi* (decimal octave low and high pitch). *kvar* can be dynamic, e.g. onset amp dependent. Pitch resolution uses the originating *spectrum ifrqs* bins/octave, with further parabolic interpolation between adjacent bins. Settings of root magnitude, *ifrqs* = 24, *iq* = 15 should capture all the inflections of interest. Between frames, the output is either repeated or interpolated at the k-rate. (See *spectrum*.)

## Examples

```

a1, a2  ins                                ; read a stereo clarinet input
krms    rms      a1, 20                    ; find a monaural rms value
kvar    =        0.6 + krms/8000          ; & use to gate the pitch var
wsig    spectrum a1, 0.01, 7, 24, 15, 0, 3 ; get a 7-oct spectrum, 24 bins
        specdisp wsig, 0.2                ; display this and now estimate
koct, ka spectrk wsig, kvar, 7.0, 10, 9, 20, 4, 0.7, 1, 5, 1, 0.2 ; the pch and amp
aosc    oscil    ka * ka * 10, cpsoct(koct), 2 ; & generate \ new tone with t
koct    =        (koct < 7.0 ? 7.0 : koct) ; replace non pitch with low C
        display koct - 7.0, 0.25, 20      ; & display the pitch track
        display ka, 0.25, 20              ; plus the summed root mag
outs    a1, aosc                            ; output 1 original and 1 new

```

# specscal

`specscal` — Scales an input spectral datablock with spectral envelopes.

## Description

Scales an input spectral datablock with spectral envelopes.

## Syntax

```
wsig specscal wsignin, ifscale, ifthresh
```

## Initialization

*ifscale* -- scale function table. A function table containing values by which a value's magnitude is rescaled.

*ifthresh* -- threshold function table. If *ifthresh* is non-zero, each magnitude is reduced by its corresponding table-value (to not less than zero)

## Performance

*wsig* -- the output spectrum

*wsignin* -- the input spectra

Scales an input spectral datablock with spectral envelopes. Function tables *ifthresh* and *ifscale* are initially sampled across the (logarithmic) frequency space of the input spectrum; then each time a new input spectrum is sensed the sampled values are used to scale each of its magnitude channels as follows: if *ifthresh* is non-zero, each magnitude is reduced by its corresponding table-value (to not less than zero); then each magnitude is rescaled by the corresponding *ifscale* value, and the resulting spectrum written to *wsig*.

## Examples

```
wsig2  specdiff  wsig1      ; sense onsets
wsig3  specfilt  wsig2, 2    ; absorb slowly
       specdisp  wsig2, 0.1  ; & display both spectra
       specdisp  wsig3, 0.1
```

## See Also

*specadm*, *specdiff*, *specfilt*, *spechist*

# specsum

specsum — Sums the magnitudes across all channels of the spectrum.

## Description

Sums the magnitudes across all channels of the spectrum.

## Syntax

```
ksum specsum wsig [, interp]
```

## Initialization

*interp* (optional, default-0) -- if non-zero, interpolate the output signal (*koct* or *ksum*). The default value is 0 (repeat the signal value between changes).

## Performance

*ksum* -- the output signal.

*wsig* -- the input spectrum.

Sums the magnitudes across all channels of the spectrum. At each new frame of *wsig*, the magnitudes are summed and released as a scalar *ksum* signal. Between frames, the output is either repeated or interpolated at the k-rate. This unit produces a k-signal summation of the magnitudes present in the spectral data, and is thereby a running measure of its moment-to-moment overall strength.

## Examples

```
ksum    specsum  wsig, 1           ; sum the spec bins, and ksmooth
        if ksum < 2000 kgoto zero   ; if sufficient amplitude
koct    specptrk wsig             ; pitch-track the signal
        kgoto    contin
zero:
koct    =         0               ; else output zero
contin:
```

## See Also

*specdisp*

# spectrum

spectrum — Generate a constant-Q, exponentially-spaced DFT.

## Description

Generate a constant-Q, exponentially-spaced DFT across all octaves of a multiply-downsampled control or audio input signal.

## Syntax

```
wsig spectrum xsig, iprd, iocts, ifrqa [, iq] [, ihann] [, idbout] \  
    [, idsprd] [, idsinrs]
```

## Initialization

*ihann* (optional) -- apply a Hamming or Hanning window to the input. The default is 0 (Hamming window)

*idbout* (optional) -- coded conversion of the DFT output:

- 0 = magnitude
- 1 = dB
- 2 = mag squared
- 3 = root magnitude

The default value is 0 (magnitude).

*idsprd* (optional) -- if non-zero, display the composite downsampling buffer every *idsprd* seconds. The default value is 0 (no display).

*idsines* (optional) -- if non-zero, display the Hamming or Hanning windowed sinusoids used in DFT filtering. The default value is 0 (no sinusoid display).

## Performance

This unit first puts signal *asig* or *ksig* through *iocts* of successive octave decimation and downsampling, and preserves a buffer of down-sampled values in each octave (optionally displayed as a composite buffer every *idsprd* seconds). Then at every *iprd* seconds, the preserved samples are passed through a filter bank (*ifrqs* parallel filters per octave, exponentially spaced, with frequency/bandwidth *Q* of *iq*), and the output magnitudes optionally converted (*idbout*) to produce a band-limited spectrum that can be read by other units.

The stages in this process are computationally intensive, and computation time varies directly with *iocts*, *ifrqs*, *iq*, and inversely with *iprd*. Settings of *ifrqs* = 12, *iq* = 10, *idbout* = 3, and *iprd* = .02 will normally be adequate, but experimentation is encouraged. *ifrqs* currently has a maximum of 120 divisions per octave. For audio input, the frequency bins are tuned to coincide with A440.

This unit produces a self-defining spectral datablock *wsig*, whose characteristics used (*iprd*, *iocts*, *ifrqs*, *idbout*) are passed via the data block itself to all derivative *wsigs*. There can be any number of *spectrum* units in an instrument or orchestra, but all *wsig* names must be unique.

## Examples

```
asig in                               ; get external audio
wsig spectrum asig, 0.02, 6, 12, 33, 0, 1, 1 ; downsample in 6 octs & calc a 72 pt dft
                                           ; (Q 33, dB out) every 20 msecs
```

# splitrig

splitrig — Split a trigger signal

## Description

*splitrig* splits a trigger signal (i.e. a timed sequence of control-rate impulses) into several channels following a structure designed by the user.

## Syntax

```
splitrig ktrig, kndx, imaxtics, ifn, kout1 [,kout2,...,koutN]
```

## Initialization

*imaxtics* - number of tics belonging to largest pattern

*ifn* - number of table containing channel-data structuring

## Performance

*asig* - incoming (input) signal

*ktrig* - trigger signal

The *splitrig* opcode splits a trigger signal into several output channels according to one or more patterns provided by the user. Normally the regular timed trigger signal generated by metro opcode is used to be transformed into rhythmic pattern that can trig several independent melodies or percussion riffs. But you can also start from non-isocronous trigger signals. This allows to use some "interpretative" and less "mechanic" groove variations. Patterns are looped and each numtics\_of\_pattern\_N the cycle is repeated.

The scheme of patterns is defined by the user and is stored into *ifn* table according to the following format:

```
g1l ftgen 1,0,1024, -2 \ ; table is generated with GEN02 in this case
\
numtics_of_pattern_1, \ ;pattern 1
  tic1_out1, tic1_out2, ... , tic1_outN,\
  tic2_out1, tic2_out2, ... , tic2_outN,\
  tic3_out1, tic3_out2, ... , tic3_outN,\
  .....
  ticN_out1, ticN_out2, ... , ticN_outN,\
\
numtics_of_pattern_2, \ ;pattern 2
  tic1_out1, tic1_out2, ... , tic1_outN,\
  tic2_out1, tic2_out2, ... , tic2_outN,\
  tic3_out1, tic3_out2, ... , tic3_outN,\
  .....
  ticN_out1, ticN_out2, ... , ticN_outN,\
  .....
\
numtics_of_pattern_N, \ ;pattern N
  tic1_out1, tic1_out2, ... , tic1_outN,\
  tic2_out1, tic2_out2, ... , tic2_outN,\
  tic3_out1, tic3_out2, ... , tic3_outN,\
  .....
```

```
ticN_out1, ticN_out2, ... , ticN_outN,\
```

This scheme can contain more than one pattern, each one with a different number of rows. Each pattern is preceded by a special row containing a single *numtics\_of\_pattern\_N* field; this field expresses the number of tics that makes up the corresponding pattern. Each pattern's row makes up a tic. Each pattern's column corresponds to a channel, and each field of a row is a number that makes up the value outputted by the corresponding *koutXX* channel (if number is a zero, corresponding output channel will not trigger anything in that particular arguments). Obviously, all rows must contain the same number of fields that must be equal to the number of *koutXX* channel. All patterns must contain the same number of rows, this number must be equal to the largest pattern and is defined by *imaxtics* variable. Even if a pattern has less tics than the largest pattern, it must be made up of the same number of rows, in this case, some of these rows, at the end of the pattern itself, will not be used (and can be set to any value, because it doesn't matter).

The *kndx* variable chooses the number of the pattern to be played, zero indicating the first pattern. Each time the integer part of *kndx* changes, tic counter is reset to zero.

Patterns are looped and each *numtics\_of\_pattern\_N* the cycle is repeated.

examples 4 - calculate average value of *asig* in the time interval

This opcode can be useful in several situations, for example to implement a vu-meter

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# sprintf

`sprintf` — printf-style formatted output to a string variable.

## Description

*sprintf* write printf-style formatted output to a string variable, similarly to the C function `sprintf()`. *sprintf* runs at i-time only.

## Syntax

```
Sdst sprintf Sfmt, xarg1[, xarg2[, ... ]]
```

## Initialization

*Sfmt* -- format string, has the same format as in `printf()` and other similar C functions, except length modifiers (l, ll, h, etc.) are not supported. The following conversion specifiers are allowed:

- d, i, o, u, x, X, e, E, f, F, g, G, c, s

*xarg1, xarg2, ...* -- input arguments (max. 30) for format, should be i-rate for all conversion specifiers except %s, which requires a string argument. Integer formats like %d round the input values to the nearest integer.

## Performance

*Sdst* -- output string variable

## Examples

Here is an example of the `sprintf` opcode. It uses the file *sprintf.csd* [examples/sprintf.csd].

### Example 912. Example of the `sprintf` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o sprintf.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

;the file "impuls20.aiff" can be found in /manual/examples
```

```
instr 1

ifn = 20
Sname sprintf "impuls%02d.aiff", ifn
Smsg sprintf "The file name is: '%s'", Sname
puts Smsg, 1
asig soundin Sname
outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 1
e
</CsScore>
</CsSoundSynthesizer>
```

Its output should include lines like this:

```
The file name is: 'impuls20.aiff'
soundin: opened 'impuls20.aiff'
```

## See Also

*sprintfk*

## Credits

Author: Istvan Varga  
2005

# sprintfk

sprintfk — printf-style formatted output to a string variable at k-rate.

## Description

*sprintfk* writes printf-style formatted output to a string variable, similarly to the C function `sprintf()`. *sprintfk* runs both at initialization and performance time.

## Syntax

```
Sdst sprintfk Sfmt, xarg1[, xarg2[, ... ]]
```

## Initialization

*Sfmt* -- format string, has the same format as in `printf()` and other similar C functions, except length modifiers (l, ll, h, etc.) are not supported. The following conversion specifiers are allowed:

- d, i, o, u, x, X, e, E, f, F, g, G, c, s

*xarg1*, *xarg2*, ... -- input arguments (max. 30) for format, should be i-rate for all conversion specifiers except %s, which requires a string argument. `sprintfk` also allows k-rate number arguments, but these should still be valid at init time as well (unless `sprintfk` is skipped with `igoto`). Integer formats like %d round the input values to the nearest integer.

## Performance

*Sdst* -- output string variable

## Examples

Here is an example of the `sprintfk` opcode. It uses the file *sprintfk.csd* [examples/sprintfk.csd].

### Example 913. Example of the `sprintfk` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o sprintfk.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr      = 48000
ksmps  = 16
nchnls = 2
0dbfs  = 1
```

```
; Example by Jonathan Murphy 2007

instr 1

S1    = "1"
S2    = " + 1"
ktrig  init    0
kval   init    2
if (ktrig == 1) then
  S1    strcatk S1, S2
  kval  = kval + 1
endif
String sprintfk "%s = %d", S1, kval
puts   String, kval
ktrig  metro    1

endin

</CsInstruments>
<CsScore>
i1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
1 + 1 = 2
1 + 1 + 1 = 3
1 + 1 + 1 + 1 = 4
1 + 1 + 1 + 1 + 1 = 5
1 + 1 + 1 + 1 + 1 + 1 = 6
1 + 1 + 1 + 1 + 1 + 1 + 1 = 7
1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 8
1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 9
1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 10
1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 11
1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 12
```

## See also

*sprintf, puts, strcat*

## Credits

Author: Istvan Varga  
2005  
Example by Jonathan Murphy

# spsend

spsend — Generates output signals based on a previously defined *space* opcode.

## Description

*spsend* depends upon the existence of a previously defined *space*. The output signals from *spsend* are derived from the values given for *xy* and *reverb* in the *space* and are ready to be sent to local or global reverb units (see example below).

## Syntax

```
a1, a2, a3, a4 spsend
```

## Performance

The configuration of the *xy* coordinates in *space* places the signal in the following way:

- a1 is -1, 1
- a2 is 1, 1
- a3 is -1, -1
- a4 is 1, -1

This assumes a loudspeaker set up as a1 is left front, a2 is right front, a3 is left back, a4 is right back. Values greater than 1 will result in sounds being attenuated, as if in the distance. *space* considers the speakers to be at a distance of 1; smaller values of *xy* can be used, but *space* will not amplify the signal in this case. It will, however balance the signal so that it can sound as if it were within the 4 speaker *space*. *x=0, y=1*, will place the signal equally balanced between left and right front channels, *x=y=0* will place the signal equally in all 4 channels, and so on. Although there must be 4 output signals from *space*, it can be used in a 2 channel orchestra. If the *xy*'s are kept so that  $Y \geq 1$ , it should work well to do panning and fixed localization in a stereo field.

## Examples

Here is a stereo example of the *spsend* opcode. It uses the file *spsend.csd* [examples/spsend.csd].

### Example 914. Example of the *spsend* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o spsend.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```
sr = 44100
ksmps = 32
Odbfs = 1
nchnls = 2 ;stereo output

ga1 init 0
ga2 init 0

instr 1 ;sends different amounts to reverb

irev = p6
asig diskin2 "fox.wav", 1
a1, a2, a3, a4 space asig, 0, 0, irev, p4, p5 ;take position values from p4, p5
ar1, ar2, ar3, ar4 spsend ;send to reverb

ga1 = ga1+ar1
ga2 = ga2+ar2
outs a1, a2

endin

instr 99 ; reverb instrument

a1 reverb2 ga1, 2.5, .5
a2 reverb2 ga2, 2.5, .5
outs a1, a2

ga1=0
ga2=0

endin

</CsInstruments>
<CsScore>
;WITH REVERB
;place the sound in the left speaker and near
i1 0 1 -1 1 .1
;place the sound in the right speaker and far
i1 1 1 45 45 .1
;place the sound equally between left and right and in the middle ground distance
i1 2 1 0 12 .1

;NO REVERB
;place the sound in the left speaker and near
i1 6 1 -1 1 0
;place the sound in the right speaker and far
i1 7 1 45 45 0
;place the sound equally between left and right and in the middle ground distance
i1 8 1 0 12 0

i 99 0 12 ;keep reverb active all the time
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*space, spdist*

## Credits

Author: Richard Karpen  
Seattle, WA USA

1998

New in Csound version 3.48

# sqrt

sqrt — Returns a square root value.

## Description

Returns the square root of  $x$  ( $x$  non-negative).

The argument value is restricted for *log*, *log10*, and *sqrt*.

## Syntax

`sqrt(x)` (no rate restriction)

`sqrt(k/i[i])` (*k*- or *i*-arrays )

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

## Examples

Here is an example of the sqrt opcode. It uses the file *sqrt.csd* [examples/sqrt.csd].

### Example 915. Example of the sqrt opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o sqrt.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 2

instr 1

asig  pluck 0.7, 55, 55, 0, 1
kpan  line 0,p3,1
kleft = sqrt(1-kpan)
kright = sqrt(kpan)
printks "square root of left channel = %f\\n", 1, kleft ;show coarse of sqaure root values
outs  asig*kleft, asig*kright      ;where 0.707126 is between 2 speakers

endin
</CsInstruments>
<CsScore>

i 1 0 10
```

```
e  
</CsScore>  
</CsoundSynthesizer>
```

Its output should include lines like these:

```
square root of left channel = 1.000000  
square root of left channel = 0.948688  
square root of left channel = 0.894437  
square root of left channel = 0.836676  
square root of left channel = 0.774620  
square root of left channel = 0.707139  
square root of left channel = 0.632499  
square root of left channel = 0.547781  
square root of left channel = 0.447295  
square root of left channel = 0.316242
```

## See Also

*abs, exp, frac, int, log, log10, i*

## sr

sr — Sets the audio sampling rate.

## Description

These statements are global value *assignments*, made at the beginning of an orchestra, before any instrument block is defined. Their function is to set certain *reserved symbol variables* that are required for performance. Once set, these reserved symbols can be used in expressions anywhere in the orchestra.

## Syntax

```
sr = iarg
```

## Initialization

*sr* = (optional) -- set sampling rate to *iarg* samples per second per channel. The default value is 44100.

In addition, any *global variable* can be initialized by an *init-time assignment* anywhere before the first *instr statement*. All of the above assignments are run as instrument 0 (i-pass only) at the start of real performance.

Beginning with Csound version 3.46, *sr* may be omitted. The sample rate will be calculated from *kr* and *ksmps*, but this must evaluate to an integer. If none of these global values is defined, the sample rate will default to 44100. You will usually want to use a value that your soundcard supports, like 44100 or 48000, otherwise, the audio generated by csound may be unplayable, or you will get an error if you attempt to run in real-time. You may naturally use a sample rate like 96000, for off-line rendering even if your soundcard doesn't support it. Csound will generate a valid file that can be played on capable systems.

## Examples

```
sr = 10000
kr = 500
ksmps = 20
gil = sr/2.
ga init 0
itranspose = octpch(.01)
```

Here is another example of the sr opcode. It uses the file *sr.csd* [examples/sr.csd].

### Example 916. Example of the sr opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac ;;realtime audio out
; -iadc ;;uncomment -iadc if real audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o sr.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;use sr to find maximum harmonics

ihar = int(sr/2/p4) ; maximum possible number of harmonics w/o aliasing
prints "maximum number of harmonics = %d \\n", ihar
kenv linen .5, 1, p3, .2 ; envelope
asig buzz kenv, p4, ihar, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
f1 0 4096 10 1 ;sine wave

i 1 0 3 100 ;different frequencies
i 1 + 3 1000
i 1 + 3 10000
e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
maximum number of harmonics = 240
maximum number of harmonics = 24
maximum number of harmonics = 2
```

## See Also

*kr, ksmps, nchnls, nchnls\_i.*

# statevar

statevar — State-variable filter.

## Description

Statevar is a new digital implementation of the analogue state-variable filter. This filter has four simultaneous outputs: high-pass, low-pass, band-pass and band-reject. This filter uses oversampling for sharper resonance (default: 3 times oversampling). It includes a resonance limiter that prevents the filter from getting unstable.

## Syntax

```
ahp,alp,abp,abr statevar ain, xcf, xq [, iosamps, istor]
```

## Initialization

*iosamps* -- number of times of oversampling used in the filtering process. This will determine the maximum sharpness of the filter resonance (Q). More oversampling allows higher Qs, less oversampling will limit the resonance. The default is 3 times (*iosamps*=0).

*istor* --initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*ahp* -- high-pass output signal.

*alp* -- low-pass output signal.

*abp* -- band-pass signal.

*abr* -- band-reject signal.

*asig* -- input signal.

*xcf* -- filter cutoff frequency (k-rate or a-rate).

*xq* -- filter Q (k-rate or a-rate). This value is limited internally depending on the frequency and the number of times of oversampling used in the process (3-times oversampling by default).

## Examples

Here is an example of the statevar opcode. It uses the file *statevar.csd* [examples/statevar.csd].

### Example 917. Example of the statevar opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o statevar.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 2

instr 1

kenv linseg 0,0.1,1, p3-0.2,1, 0.1, 0 ;declick envelope
asig buzz .6*kenv, 100, 100, 1
kf expseg 100, p3/2, 5000, p3/2, 1000 ;envelope for filter cutoff
ahp,alp,abp,abr statevar asig, kf, 4
outs alp,ahp ; lowpass left, highpass right

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine wave

i1 0 5
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
January 2005

New plugin in version 5

January 2005.

Audio rate parameters introduced in version 6.02

October 2013.

# stix

stix — Semi-physical model of a stick sound.

## Description

*stix* is a semi-physical model of a stick sound. It is one of the PhISEM percussion opcodes. PhISEM (Physically Informed Stochastic Event Modeling) is an algorithmic approach for simulating collisions of multiple independent sound producing objects.

## Syntax

```
ares stix iamp, idettack [, inum] [, idamp] [, imaxshake]
```

## Initialization

*iamp* -- Amplitude of output. Note: As these instruments are stochastic, this is only a approximation.

*idettack* -- period of time over which all sound is stopped

*inum* (optional) -- The number of beads, teeth, bells, timbrels, etc. If zero, the default value is 30.

*idamp* (optional) -- the damping factor, as part of this equation:

$$\text{damping\_amount} = 0.998 + (\text{idamp} * 0.002)$$

The default *damping\_amount* is 0.998 which means that the default value of *idamp* is 0. The maximum *damping\_amount* is 1.0 (no damping). This means the maximum value for *idamp* is 1.0.

The recommended range for *idamp* is usually below 75% of the maximum value.

*imaxshake* (optional) -- amount of energy to add back into the system. The value should be in range 0 to 1.

## Examples

Here is an example of the stix opcode. It uses the file *stix.csd* [examples/stix.csd].

### Example 918. Example of the stix opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o stix.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
```

```
ksmps = 32
0dbfs = 1
nchnls = 2

instr 1

idamp = p4 ;vary damping amount
asig stix .5, 0.01, 30, idamp
outs asig, asig

endin
</CsInstruments>
<CsScore>

i1 0 1 .3
i1 + 1 >
i1 + 1 >
i1 + 1 .95

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*cabasa, crunch, sandpaper, sekere*

## Credits

Author: Perry Cook, part of the PhOLIES (Physically-Oriented Library of Imitated Environmental Sounds)  
Adapted by John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 4.07

Added notes by Rasmus Ekman on May 2002.

# STKBandedWG

STKBandedWG — STKBandedWG uses banded waveguide techniques to model a variety of sounds.

## Description

This opcode uses banded waveguide techniques to model a variety of sounds, including bowed bars, glasses, and bowls.

## Syntax

```
asignal STKBandedWG ifrequency, iamplitude, [kpress, kv1[, kmot, kv2[, klfo, kv3[, klfodepth, kv4[, kv5[, kv6[, kv7[, kv8]
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*kpress* -- controller 2, pressure of bow. Value range of kv1 is 0-127.

*kmot* -- controller 4, motion of bow. Value range of kv2 is 0-127.

*klfo* -- controller 11, speed of low-frequency oscillator. Value range of kv3 is 0-127.

*klfodepth* -- controller 1, depth of low-frequency oscillator. Value range of kv4 is 0-127.

*kvel* -- controller 128, velocity of bow. Value range of kv5 is 0-127.

*kstrk* -- controller 64, striking of bow. Value range of kv6 is 0-127.

*kinstr* -- controller 16, instrument presets (0 = uniform bar, 1 = tuned bar, 2 = glass harmonica, 3 = Tibetan bowl). Value range of kv7 is 0-3.



### Notes

The code for this opcode is taken directly from the BandedWG class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the controller values are krate variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKBandedWG opcode. It uses the file *STKBandedWG.csd* [examples/STKBandedWG.csd].

**Example 919. Example of the STKBandedWG opcode.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKBandedWG.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifrq =   p4
kv1 line p5, p3, p6      ;pressure of bow
kenv line 1, p3, 0

asig STKBandedWG cpspch(ifrq), 1, 2, kv1, 4, 100, 11, 0, 1, 0, 64, 100, 128, 120, 16, 2
asig = asig * kenv      ;simple envelope
outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 10 5.00 100 0
i 1 10 8 6.03 10 .
i 1 20 5 7.05 50 127

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Michael Gogins (after Georg Essl)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKBeeThree

STKBeeThree — STK Hammond-oid organ-like FM synthesis instrument.

## Description

STK Hammond-oid organ-like FM synthesis instrument.

This opcode a simple 4 operator topology, also referred to as algorithm 8 of the TX81Z. It simulates the sound of a Hammond-oid organ, and some related sounds.

## Syntax

```
asignal STKBeeThree ifrequency, iamplitude, [kop4, kv1[, kop3, kv2[, klfo, kv3[, klfodepth, kv4[, kadsr
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*kop4* -- controller 2, gain of feedback of operator 4. Value range of kv1 is 0-127.

*kop3* -- controller 4, gain of operator 3. Value range of kv2 is 0-127.

*klfo* -- controller 11, speed of low-frequency oscillator. Value range of kv3 is 0-127.

*klfodepth* -- controller 1, depth of low-frequency oscillator. Value range of kv4 is 0-127.

*kadsr* -- controller 128, ADSR 2 and 4 target. Value range of kv5 is 0-127.



### Notes

The code for this opcode is taken directly from the BeeThree class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the controller values are krate variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKBeeThree opcode. It uses the file *STKBeeThree.csd* [examples/STKBeeThree.csd].

**Example 920. Example of the STKBeeThree opcode.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKBeeThree.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kv1 = p6      ;feedback of operator 4
kv2 line p4, p3, p5 ;gain of operator 3
kv5 line 0, p3, 100
ipch = p7

asig STKBeeThree cpspch(ipch), 1, 2, kv1, 4, kv2, 11, 50, 1, 0, 128, kv5
outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 2 20 100 127 8.00
i 1 + 8 120 0 0 6.09
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Michael Gogins (after Perry Cook)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKBlowBotl

STKBlowBotl — STKBlowBotl uses a helmholtz resonator (biquad filter) with a polynomial jet excitation.

## Description

This opcode implements a helmholtz resonator (biquad filter) with a polynomial jet excitation (a la Cook).

## Syntax

```
asignal STKBlowBotl ifrequency, iamplitude, [knoise, kv1[, klfo, kv2[, klfodepth, kv3[, kvol, kv4]]]]
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*knoise* -- controller 4, gain of noise. Value range of kv1 is 0-127.

*klfo* -- controller 11, speed of low-frequency oscillator. Value range of kv2 is 0-127.

*klfodepth* -- controller 1, depth of low-frequency oscillator. Value range of kv3 is 0-127.

*kvol* -- controller 128, volume. Value range of kv4 is 0-127.



### Notes

The code for this opcode is taken directly from the BlowBotl class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the controller values are krate variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKBlowBotl opcode. It uses the file *STKBlowBotl.csd* [examples/STKBlowBotl.csd].

### Example 921. Example of the STKBlowBotl opcode.

```
<CsoundSynthesizer>
```

```
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKBlowBotl.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ipch = p4
kv1 line p5, p3, p6      ;gain of noise
kv4 line 100, p3, 70    ;volume

asig STKBlowBotl cpspch(ipch), 1, 4, kv1, 11, 10, 1, 50, 128, kv4
asig = asig * .5        ;too loud
outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 2 9.00 20 100
i 1 + 5 8.03 120 0
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Michael Gogins (after Perry Cook)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKBlowHole

STKBlowHole — STK clarinet physical model with one register hole and one tonehole.

## Description

This opcode is based on the clarinet model, with the addition of a two-port register hole and a three-port dynamic tonehole implementation.

In this implementation, the distances between the reed/register hole and tonehole/bell are fixed. As a result, both the tonehole and register hole will have variable influence on the playing frequency, which is dependent on the length of the air column. In addition, the highest playing frequency is limited by these fixed lengths.

## Syntax

```
asignal STKBlowHole ifrequency, iamplitude, [kreed, kv1[, knoise, kv2[, khole, kv3[, kreg, kv4[, kbreath
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*kreed* -- controller 2, stiffness of reed. Value range of kv1 is 0-127.

*knoise* -- controller 4, gain of noise. Value range of kv2 is 0-127.

*khole* -- controller 11, state of tonehole. Value range of kv3 is 0-127.

*kreg* -- controller 1, state of register. Value range of kv4 is 0-127.

*kbreath* -- controller 128, breath pressure. Value range of kv5 is 0-127.



### Notes

The code for this opcode is taken directly from the BlowHole class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the controller values are krate variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKBlowHole opcode. It uses the file *STKBlowHole.csd* [examples/STKBlowHole.csd].

**Example 922. Example of the STKBlowHole opcode.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKBlowHole.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ipch = p4
kv1 = p7      ;stiffness of reed
kv3 line p5, p3, p6      ;state of tonehole

asig STKBlowHole cpspch(ipch), 1, 2, kv1, 4, 100, 11, kv3, 1, 10, 128, 100
outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 4 10.00 20 127 100
i 1 + 7 6.09 120 0 10
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Michael Gogins (after Perry Cook)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKBowed

STKBowed — STKBowed is a bowed string instrument.

## Description

STKBowed is a bowed string instrument, using a waveguide model.

## Syntax

```
asignal STKBowed ifrequency, iamplitude, [kpress, kv1[, kpos, kv2[, klfo, kv3[, klfodepth, kv4[, kvol,
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*kpress* -- controller 2, bow pressure. Value range of kv1 is 0-127.

*kpos* -- controller 4, position on bow. Value range of kv2 is 0-127.

*klfo* -- controller 11, speed of low-frequency oscillator. Value range of kv3 is 0-127.

*klfodepth* -- controller 1, depth of low-frequency oscillator. Value range of kv4 is 0-127.

*kvol* -- controller 128, volume. Value range of kv5 is 0-127.



### Note

The code for this opcode is taken directly from the Bowed class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the controller values are krate variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKBowed opcode. It uses the file *STKBowed.csd* [examples/STKBowed.csd].

**Example 923. Example of the STKBowed opcode.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKBowed.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ipch = p4
kv2 = p7      ;position on bow
kv1 line p5, p3, p6      ;bow pressure
kv4 line 0, p3, 7      ;depth of low-frequency oscillator

asig STKBowed cpspch(ipch), 1, 2, kv1, 4, kv2, 11, 40, 1, kv4, 128, 100
asig = asig*2      ;amplify
outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 15 6.00 20 100 115
i 1 17 3 7.00 120 0 0
i 1 21 3 7.09 120 0 30
i 1 21 4 7.03 50 0 0
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Michael Gogins (after Perry Cook)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKBrass

STKBrass — STKBrass is a simple brass instrument.

## Description

STKBrass uses a simple brass instrument waveguide model, a la Cook.

## Syntax

```
asignal STKBrass ifrequency, iamplitude, [klip, kv1[, kslide, kv2[, klfo, kv3[, klfodepth, kv4[, kvol,
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*klip* -- controller 2, lip tension. Value range of kv1 is 0-127.

*kslide* -- controller 4, slide length. Value range of kv2 is 0-127.

*klfo* -- controller 11, speed of low-frequency oscillator. Value range of kv3 is 0-127.

*klfodepth* -- controller 1, depth of low-frequency oscillator. Value range of kv4 is 0-127.

*kvol* -- controller 128, volume. Value range of kv5 is 0-127.



### Note

The code for this opcode is taken directly from the Brass class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the controller values are krate variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKBrass opcode. It uses the file *STKBrass.csd* [examples/STKBrass.csd].

**Example 924. Example of the STKBrass opcode.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKBrass.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifrq = p4
kv1 line p5, p3, p6      ;lip tension
kv3 line p7, p3, p8     ;speed of low-frequency oscillator

asig STKBrass cpspch(ifrq), 1, 2, kv1, 4, 100, 11, kv3, 1, 10, 128, 40
asig = asig *3          ;amplify
outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 2 8.05 100 120 50 0
i 1 + 3 9.00 80 82 10 0
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Michael Gogins (after Perry Cook)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKClarinet

STKClarinet — STKClarinet uses a simple clarinet physical model.

## Description

STKClarinet uses a simple clarinet physical model.

## Syntax

```
asignal STKClarinet ifrequency, iamplitude, [kstiff, kv1[, knoise, kv2[, klfo, kv3[, klfodepth, kv4[, k
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*kstiff* -- controller 2, reed stiffness. Value range of kv1 is 0-127.

*knoise* -- controller 4, gain of noise. Value range of kv2 is 0-127.

*klfo* -- controller 11, speed of low-frequency oscillator. Value range of kv3 is 0-127.

*klfodepth* -- controller 1, depth of low-frequency oscillator. Value range of kv4 is 0-127.

*kbreath* -- controller 128, breath pressure. Value range of kv5 is 0-127.



### Note

The code for this opcode is taken directly from the Clarinet class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the controller values are krate variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKClarinet opcode. It uses the file *STKClarinet.csd* [examples/STKClarinet.csd].

### Example 925. Example of the STKClarinet opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o STKclarinet.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifrq = p4
kv5 = p5      ;breath pressure
kv1 line p6, p3, p7      ;reed stiffness
asig STKClarinet cpspch(p4), 1, 2, kv1, 4, 100, 11, 60, 1, 10, 128, kv5
outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 3 8.00 100 127 10
i 1 + 10 8.08 80 60 100
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*STKFlute*.

## Credits

Author: Michael Gogins (after Perry Cook)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKDrummer

STKDrummer — STKDrummer is a drum sampling synthesizer.

## Description

STKDrummer is a drum sampling synthesizer using raw waves and one-pole filters, The drum rawwave files are sampled at 22050 Hz, but will be appropriately interpolated for other sample rates.

## Syntax

```
asignal STKDrummer ifrequency, iamplitude
```

## Initialization

*ifrequency* -- Samples being played.

*iamplitude* -- Amplitude of note played (range 0-1).



### Note

The code for this opcode is taken directly from the Drummer class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

## Examples

Here is an example of the STKDrummer opcode. It uses the file *STKDrummer.csd* [examples/STKDrummer.csd].

### Example 926. Example of the STKDrummer opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      -M0   ;;realtime audio out and midi in
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKDrummer.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;STK Drummer - has no controllers but plays samples (11)

icps cpsmidi
iamp ampmidi 1
asig STKDrummer icps, iamp
outs asig, asig
endin
```

```
</CsInstruments>
<CsScore>
; play 5 minutes
f0 300
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*STKClarinet.*

## Credits

Author: Michael Gogins (after Perry Cook)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKFlute

STKFlute — STKFlute uses a simple flute physical model.

## Description

STKFlute uses a simple flute physical model. The jet model uses a polynomial, a la Cook.

## Syntax

```
asignal STKFlute ifrequency, iamplitude, [kjet, kv1[, knoise, kv2[, klfo, kv3[, klfodepth, kv4[, kbreath
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*kjet* -- controller 2, jet delay. Value range of kv1 is 0-127.

*knoise* -- controller 4, gain of noise. Value range of kv2 is 0-127.

*klfo* -- controller 11, speed of low-frequency oscillator. Value range of kv3 is 0-127.

*klfodepth* -- controller 1, depth of low-frequency oscillator. Value range of kv4 is 0-127.

*kbreath* -- controller 128, breath pressure. Value range of kv5 is 0-127.



### Note

The code for this opcode is taken directly from the Flute class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the controller values are krate variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKFlute opcode. It uses the file *STKFlute.csd* [examples/STKFlute.csd].

**Example 927. Example of the STKFlute opcode.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o STKFlute.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifrq = p4
kv1 line p5, p3, p6      ;jet delay
kv4 line 0, p3, 100     ;vibrato depth

asig STKFlute cpspch(ifrq), 1, 2, kv1, 4, 100, 11, 100, 1, kv4, 128, 100
outs asig, asig
endin

</CsInstruments>
<CsScore>
i 1 0 5 8.00 0 0
i 1 7 3 9.00 20 120
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*STKClarinete*.

## Credits

Author: Michael Gogins (after Perry Cook)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKFMVoices

STKFMVoices — STKFMVoices is a singing FM synthesis instrument.

## Description

STKFMVoices is a singing FM synthesis instrument. It has 3 carriers and a common modulator, also referred to as algorithm 6 of the TX81Z.

## Syntax

```
asignal STKFMVoices ifrequency, iamplitude, [kvowel, kv1[, kspec, kv2[, klfo, kv3[, klfodepth, kv4[, ka
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*kvowel* -- controller 2, vowel. Value range of kv1 is 0-127.

*kspec* -- controller 4, spectral tilt. Value range of kv2 is 0-127.

*klfo* -- controller 11, speed of low-frequency oscillator. Value range of kv3 is 0-127.

*klfodepth* -- controller 1, depth of low-frequency oscillator. Value range of kv4 is 0-127.

*kadsr* -- controller 128, ADSR 2 and 4 Target. Value range of kv5 is 0-127.



### Note

The code for this opcode is taken directly from the FMVoices class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the controller values are krate variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKFMVoices opcode. It uses the file *STKFMVoices.csd* [examples/STKFMVoices.csd].

**Example 928. Example of the STKFMVoices opcode.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKFMVoices.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifrq = p4
kv1 line p5, p3, p6      ;vowel
kv2 line p7, p3, p8     ;specral tilt

asig STKFMVoices cspch(ifrq), 1, 2, kv1, 4, kv2, 11, 10, 1, 10, 128, 50
asig = asig * 4         ;amplify
outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 5 5.00 10 120 0 0
i 1 + 2 8.00 80 82 127 0
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*STKBeeThree.*

## Credits

Author: Michael Gogins (after Perry Cook)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKHevyMetl

STKHevyMetl — STKHevyMetl produces metal sounds.

## Description

STKHevyMetl produces metal sounds, using FM synthesis. It uses 3 cascade operators with feedback modulation, also referred to as algorithm 3 of the TX81Z.

## Syntax

```
asignal STKHevyMetl ifrequency, iamplitude, [kmod, kv1[, kcross, kv2[, klfo, kv3[, klfodepth, kv4[, ka
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*kmod* -- controller2, total modulator index. Value range of kv1 is 0-127.

*kcross* -- controller 4, crossfade of modulator. Value range of kv2 is 0-127.

*klfo* -- controller 11, speed of low-frequency oscillator. Value range of kv3 is 0-127.

*klfodepth* -- controller 1, depth of low-frequency oscillator. Value range of kv4 is 0-127.

*kadsr* -- controller 128, ADSR 2 and 4 target. Value range of kv5 is 0-127.



### Note

The code for this opcode is taken directly from the HevyMetl class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the controller values are krate variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKHevyMetl opcode. It uses the file *STKHevyMetl.csd* [examples/STKHevyMetl.csd].

**Example 929. Example of the STKHevyMetl opcode.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKHevyMet1.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifrq = p4
kv1 line p5, p3, p6      ;Total Modulator Index
kv2 line p7, p3, 0      ;Modulator Crossfade

asig STKHevyMet1 cspch(ifrq), 1, 2, kv1, 4, kv2, 11, 0, 1, 100, 128, 40
outs asig, asig
endin

</CsInstruments>
<CsScore>
i 1 0 7 8.05 100 0 100
i 1 3 7 9.03 20 120 0
i 1 3 .5 8.05 20 120 0
i 1 4 .5 9.09 20 120 0
i 1 5 3 9.00 20 120 0

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*STKWurley.*

## Credits

Author: Michael Gogins (after Perry Cook)

Irreducible Productions

New York, NY

New in Csound version 5.11

# STKMandolin

STKMandolin — STKMandolin produces mandolin-like sounds.

## Description

STKMandolin produces mandolin-like sounds, using "commuted synthesis" techniques to model a mandolin instrument.

## Syntax

```
asignal STKMandolin ifrequency, iamplitude, [kbody, kv1[, kpos, kv2[, ksus, kv3[, kdetune, kv4[, kmic,
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*kbody* -- controller 2, size of body. Value range of kv1 is 0-127.

*kpos* -- controller 4, pluck position. Value range of kv2 is 0-127.

*ksus* -- controller 11, string sustain. Value range of kv3 is 0-127.

*kdetune* -- controller 1, string detuning. Value range of kv4 is 0-127.

*kmic* -- controller 128, position of microphone. Value range of kv5 is 0-127.



### Note

The code for this opcode is taken directly from the Mandolin class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the controller values are krate variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKMandolin opcode. It uses the file *STKMandolin.csd* [examples/STKMandolin.csd].

**Example 930. Example of the STKMandolin opcode.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKMandolin.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifrq = p4
kv1 line p5, p3, p6 ;body size
kv3 = p7 ;sustain

asig STKMandolin cspch(ifrq), 1, 2, kv1, 4, 10, 11, kv3, 1, 100, 128, 100
outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 .3 7.00 100 0 20
i 1 + . 8.00 10 100 20
i 1 + . 8.00 100 0 120
i 1 + 4 8.00 10 10 127
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*STKPlucked.*

## Credits

Author: Michael Gogins (after Perry Cook)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKModalBar

STKModalBar — STKModalBar is a resonant bar instrument.

## Description

This opcode is a resonant bar instrument. It has a number of different struck bar instruments.

## Syntax

```
asignal STKModalBar ifrequency, iamplitude, [khard, kv1[, kpos, kv2[, klfo, kv3[, klfodepth, kv4[, kmix
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*khard* -- controller 2, hardness of the stick. Value range of kv1 is 0-127.

*kpos* -- controller 4, stick position. Value range of kv2 is 0-127.

*klfo* -- controller 11, speed of low-frequency oscillator. Value range of kv3 is 0-127.

*klfodepth* -- controller 1, depth of low-frequency oscillator. Value range of kv4 is 0-127.

*kmix* -- controller 8, direct stick mix. Value range of kv5 is 0-127.

*kvol* -- controller 128, volume. Value range of kv6 is 0-127.

*kinstr* -- controller 16, instrument presets (0 = marimba, 1 = vibraphone, 2 = agogo, 3 = wood1, 4 = reso, 5 = wood2, 6 = beats, 7 = two fixed, 8 = clump). Value range of kv7 is 0-16.



### Note

The code for this opcode is taken directly from the ModalBar class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the controller values are krate variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKModalBar opcode. It uses the file *STKModalBar.csd* [examples/STKModalBar.csd].

**Example 931. Example of the STKModalBar opcode.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;RT audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKModalBar.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifrq = p4
kv1 = p5      ;stick hardness      ;

asig STKModalBar cpspch(ifrq), 1, 2, kv1, 4, 120, 11, 0, 1, 0, 8, 10, 16, 1
asig = asig * 3      ;amplify
outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 2 8.00 0
i 1 + 2 8.05 120
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Michael Gogins (after Georg Essl)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKMoog

STKMoog — STKMoog produces moog-like swept filter sounds.

## Description

STKMoog produces moog-like swept filter sounds, using one attack wave, one looped wave, and an ADSR envelope and adds two sweepable formant filters.

## Syntax

```
asignal STKMoog ifrequency, iamplitude, [kq, kv1[, krate, kv2[, klfo, kv3[, klfodepth, kv4[, kvol, kv5]
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*kq* -- controller 2, Q filter. Value range of kv1 is 0-127.

*krate* -- controller 4, rate of filter sweep. Value range of kv2 is 0-127.

*klfo* -- controller 11, speed of low-frequency oscillator. Value range of kv3 is 0-127.

*klfodepth* -- controller 1, depth of low-frequency oscillator. Value range of kv4 is 0-127.

*kvol* -- controller 128, volume. Value range of kv5 is 0-127.



### Note

The code for this opcode is taken directly from the Moog class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the controller values are krate variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKMoog opcode. It uses the file *STKMoog.csd* [examples/STKMoog.csd].

### Example 932. Example of the STKMoog opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKMoog.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifrq = p4
kv1 line p5, p3, p6      ;filter Q

asig STKMoog cpspch(ifrq), 1, 2,kv1, 4, 120, 11, 40, 1, 1, 128, 120
asig = asig * .3      ;too loud
outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 .5 6.00 100 0
i 1 + . 5.05 10 127
i 1 + . 7.06 100 0
i 1 + 3 7.00 10 10
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Michael Gogins (after Perry Cook)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKPercFlut

STKPercFlut — STKPercFlut is a percussive flute FM synthesis instrument.

## Description

STKPercFlut is a percussive flute FM synthesis instrument. The instrument uses an algorithm like the algorithm 4 of the TX81Z.

## Syntax

```
asignal STKPercFlut ifrequency, iamplitude, [kmod, kv1[, kcross, kv2[, klfo, kv3[, klfodepth, kv4[, ka
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*kmod* -- controller 2, total modulator index. Value range of kv1 is 0-127.

*kcross* -- controller 4, crossfade of modulator. Value range of kv2 is 0-127.

*klfo* -- controller 11, speed of low-frequency oscillator. Value range of kv3 is 0-127.

*klfodepth* -- controller 1, depth of low-frequency oscillator. Value range of kv4 is 0-127.

*kadsr* -- controller 128, ADSR 2 and 4 target. Value range of kv5 is 0-127.



### Note

The code for this opcode is taken directly from the PercFlut class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the controller values are krate variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKPercFlut opcode. It uses the file *STKPercFlut.csd* [examples/STKPercFlut.csd].

**Example 933. Example of the STKPercFlut opcode.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKPercFlut.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifrq = p4
kv1 line p5, p3, p6      ;Total Modulator Index
kv2 line p7, p3, 0      ;Modulator Crossfade

asig STKPercFlut cspch(ifrq), 1, 2, kv1, 4, kv2, 11, 0, 1, 100, 128, 40
outs asig, asig
endin

</CsInstruments>
<CsScore>
i 1 0 7 8.05 100 0 100
i 1 3 7 9.03 20 120 0
i 1 3 .5 8.05 20 120 0
i 1 4 .5 9.09 20 120 0
i 1 5 3 9.00 20 120 0

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Michael Gogins (after Perry Cook)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKPlucked

STKPlucked — STKPlucked uses a plucked string physical model.

## Description

STKPlucked uses a plucked string physical model based on the Karplus-Strong algorithm.

## Syntax

```
asignal STKPlucked ifrequency, iamplitude
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).



### Note

The code for this opcode is taken directly from the Plucked class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

## Examples

Here is an example of the STKPlucked opcode. It uses the file *STKPlucked.csd* [examples/STKPlucked.csd].

### Example 934. Example of the STKPlucked opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKPlucked.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;STKPlucked - has no controllers

ifrq = p4

asig STKPlucked cpspch(ifrq), 1
outs asig, asig
endin
```

```
</CsInstruments>
<CsScore>

i 1 0 2 6.00
i 1 + 8 5.00
i 1 + .5 8.00
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*STKSitar*.

## Credits

Author: Michael Gogins (after Perry Cook)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKResonate

STKResonate — STKResonate is a noise driven formant filter.

## Description

STKResonate is a noise driven formant filter. This instrument contains a noise source, which excites a biquad resonance filter, with volume controlled by an ADSR.

## Syntax

```
asignal STKResonate ifrequency, iamplitude, [kfreq, kv1[, kpole, kv2[, knotch, kv3[, kzzero, kv4[, kenv,
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*kfreq* -- controller 2, frequency of resonance. Value range of kv1 is 0-127.

*kpole* -- controller 4, pole radii. Value range of kv2 is 0-127.

*knotch* -- controller 11, notch frequency. Value range of kv3 is 0-127.

*kzero* -- controller 1, zero radii. Value range of kv4 is 0-127.

*kenv* -- controller 128, gain of envelope. Value range of kv5 is 0-127.



### Note

The code for this opcode is taken directly from the Resonate class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the controller values are krate variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKResonate opcode. It uses the file *STKResonate.csd* [examples/STKResonate.csd].

**Example 935. Example of the STKResonate opcode.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKResonate.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; frequency and amplitude of STKResonate have no effect

kv2 = p4      ;pole radii
kv1 line 100, p3, 0 ;resonance freq + notch freq
kv3 = kv1
asig STKResonate 1, 1, 2, kv1, 4, kv2, 1, 10, 11, kv3, 128, 100
asig = asig * .3 ;too loud
outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 1 0
i 1 + . >
i 1 + . >
i 1 + . >
i 1 + . 120
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Michael Gogins (after Perry Cook)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKRhodey

STKRhodey — STK Fender Rhodes-like electric piano FM synthesis instrument.

## Description

STK Fender Rhodes-like electric piano FM synthesis instrument.

This opcode implements an instrument based on two simple FM Pairs summed together, also referred to as algorithm 5 of the Yamaha TX81Z. It simulates the sound of a Rhodes electric piano, and some related sounds.

## Syntax

```
asignal STKRhodey ifrequency, iamplitude, [kmod, kv1[, kcross, kv2[, klfo, kv3[, klfodepth, kv4[, kadsr
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*kmod* -- controller 2, modulator index 1. Value range of kv1 is 0-127.

*kcross* -- controller 4, crossfade of outputs. Value range of kv2 is 0-127.

*klfo* -- controller 11, speed of low-frequency oscillator. Value range of kv3 is 0-127.

*klfodepth* -- controller1, depth of low-frequency oscillator. Value range of kv4 is 0-127.

*kadsr* -- controller 128, ADSR 2 and 4 target. Value range of kv5 is 0-127.



### Note

The code for this opcode is taken directly from the Rhodey class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the controller values are krate variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKRhodey opcode. It uses the file *STKRhodey.csd* [examples/STKRhodey.csd].

**Example 936. Example of the STKRhodey opcode.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKRhodey.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifrq = p4
kv1 line p5, p3, p6      ;(FM) Modulator Index One
kv5 = p7                ;ADSR 2 and 4 target
asig STKRhodey cpspch(p4), 1, 2, kv1, 4, 10, 11, 100, 1, 3, 128, kv5
  outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 .5 7.00 75 0 0
i 1 + . 8.00 120 0 120
i 1 + 1 6.00 50 120 50
i 1 + 4 8.00 10 120 100
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*STKWurley.*

## Credits

Author: Michael Gogins (after Perry Cook)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKSaxofony

STKSaxofony — STKSaxofony is a faux conical bore reed instrument.

## Description

STKSaxofony is a faux conical bore reed instrument. This opcode uses a "hybrid" digital waveguide instrument that can generate a variety of wind-like sounds. It has also been referred to as the "blowed string" model. The waveguide section is essentially that of a string, with one rigid and one lossy termination. The non-linear function is a reed table. The string can be "blown" at any point between the terminations, though just as with strings, it is impossible to excite the system at either end. If the excitation is placed at the string mid-point, the sound is that of a clarinet. At points closer to the "bridge", the sound is closer to that of a saxophone.

## Syntax

```
asignal STKSaxofony ifrequency, iamplitude, [kstiff, kv1[, kapert, kv2[, kblow, kv3[, knoise, kv4[, klf
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*kstiff* -- controller 2, stiffness of reed. Value range of kv1 is 0-127.

*kapert* -- controller 26, reed aperture. Value range of kv2 is 0-127.

*kblow* -- controller 11, blow position. Value range of kv3 is 0-127.

*knoise* -- controller 4, noise gain. Value range of kv4 is 0-127.

*klfo* -- controller 29, speed of low-frequency oscillator. Value range of kv5 is 0-127.

*klfodepth* -- controller 1, depth of low-frequency oscillator. Value range of kv6 is 0-127.

*kbreath* -- controller 128, breath pressure. Value range of kv7 is 0-127.



### Note

The code for this opcode is taken directly from the Saxofony class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the controller values are krate variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The

order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKSaxofony opcode. It uses the file *STKSaxofony.csd* [examples/STKSaxofony.csd].

### Example 937. Example of the STKSaxofony opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKSaxofony.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifreq = p4
kv1 = p5      ;reed stiffness
kv3 line p6, p3, p7      ;blow position
kv6 line 0, p3, 127      ;depth of low-frequency oscillator

asig STKSaxofony cpspch(p4), 1, 2, kv1, 4, 100, 26, 70, 11, kv3, 1, kv6, 29, 100
asig = asig * .5      ;too loud
outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 3 6.00 30 100 10
i 1 + . 8.00 30 100 100
i 1 + . 7.00 90 127 30
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Michael Gogins (after Georg Essl)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKShakers

STKShakers — STKShakers is an instrument that simulates environmental sounds or collisions of multiple independent sound producing objects.

## Description

STKShakers are a set of PhISEM and PhOLIES instruments: PhISEM (Physically Informed Stochastic Event Modeling) is an algorithmic approach for simulating collisions of multiple independent sound producing objects. It can simulate a Maraca, Sekere, Cabasa, Bamboo Wind Chimes, Water Drops, Tambourine, Sleighbells, and a Guiro. On <http://soundlab.cs.princeton.edu/research/controllers/shakers/> PhOLIES (Physically-Oriented Library of Imitated Environmental Sounds) there is a similar approach for the synthesis of environmental sounds. It simulates of breaking sticks, crunchy snow (or not), a wrench, sandpaper, and more..

## Syntax

```
asignal STKShakers ifrequency, iamplitude, [kenerg, kv1[, kdecay, kv2[, kshake, kv3[, knum, kv4[, kres,
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*kenerg* -- controller 2, shake energy. Value range of kv1 is 0-127.

*kdecay* -- controller 4, system decay. Value range of kv2 is 0-127.

*kshake* -- controller 128, shake energy. Value range of kv3 is 0-127.

*knum* -- controller 11, number of objects. Value range of kv4 is 0-127.

*kres* -- controller 1, resonance frequency. Value range of kv5 is 0-127.

*kinstr* -- controller 1071, instrument selection (Maraca = 0, Cabasa = 1, Sekere = 2, Guiro = 3, Water Drops = 4, Bamboo Chimes = 5, Tambourine = 6, Sleigh Bells = 7, Sticks = 8, Crunch = 9, Wrench = 10, Sand Paper = 11, Coke Can = 12, Next Mug = 13, Penny + Mug = 14, Nickle + Mug = 15, Dime + Mug = 16, Quarter + Mug = 17, Franc + Mug = 18, Peso + Mug = 19, Big Rocks = 20, Little Rocks = 21, Tuned Bamboo Chimes = 22). Value range of kv6 is 0-22.



### Note

The code for this opcode is taken directly from the Shakers class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the con-

troller values are krte variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKShakers opcode. It uses the file *STKShakerscsd* [examples/STKShakers.csd].

### Example 938. Example of the STKShakers opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKShakers.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifrq = p4

asig STKShakers cpspch(p4), 1, 2, 10, 4, 10, 11, 10, 1, 112, 128, 80, 1071, 5
asig = asig *2      ;amplify
outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0.2 .5 7.00 75 0 20

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Michael Gogins (after Georg Essl)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKSimple

STKSimple — STKSimple is a wavetable/noise instrument.

## Description

STKSimple is a wavetable/noise instrument. It combines a looped wave, a noise source, a biquad resonance filter, a one-pole filter, and an ADSR envelope to create some interesting sounds.

## Syntax

```
asignal STKSimple ifrequency, iamplitude, [kpos, kv1[, kcross, kv2[, kenv, kv3[, kgain, kv4]]]]
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*kpos* -- controller 2, position of filter pole. Value range of kv1 is 0-127.

*kcross* -- controller 4, noise/pitched cross-fade. Value range of kv2 is 0-127.

*kenv* -- controller 11, rate of envelope. Value range of kv3 is 0-127.

*kgain* -- controller 128, gain. Value range of kv4 is 0-127.



### Note

The code for this opcode is taken directly from the Simple class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the controller values are krate variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKSimple opcode. It uses the file *STKSimple.csd* [examples/STKSimple.csd].

### Example 939. Example of the STKSimple opcode.

```
<CsoundSynthesizer>
```

```
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKSimple.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifrq = p4
kv1 line p5, p3, p6      ;Filter Pole Position
kv2 line 20, p3, 90     ;Noise/Pitched Cross-Fade

asig STKSimple cpspch(p4), 1, 2, kv1, 4, kv2, 11, 10, 128, 120
outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 5 7.00 100 0
i 1 + . 7.05 10 127
i 1 + . 8.03 100 0
i 1 + . 5.00 10 10
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*STKClarinete*.

## Credits

Author: Michael Gogins (after Perry Cook)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKSitar

STKSitar — STKSitar uses a plucked string physical model.

## Description

STKSitar uses a plucked string physical model based on the Karplus-Strong algorithm.

## Syntax

```
asignal STKSitar ifrequency, iamplitude
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).



### Note

The code for this opcode is taken directly from the Sitar class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

## Examples

Here is an example of the STKSitar opcode. It uses the file *STKSitar.csd* [examples/STKSitar.csd].

### Example 940. Example of the STKSitar opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKSitar.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;STKSitar - has no controllers

ifrq = p4

asig STKSitar cpspch(p4), 1
asig = asig * 2 ;amplify
outs asig, asig
endin
```

```
</CsInstruments>
<CsScore>

i 1 0 4 6.00
i 1 + 2 7.05
i 1 + 7 5.05
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Michael Gogins (after Perry Cook)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKStifKarp

STKStifKarp — STKStifKarp is a plucked stiff string instrument.

## Description

STKStifKarp is a plucked stiff string instrument. It a simple plucked string algorithm (Karplus Strong) with enhancements, including string stiffness and pluck position controls. The stiffness is modeled with allpass filters.

## Syntax

```
asignal STKStifKarp ifrequency, iamplitude, [kpos, kv1[, ksus, kv2[, kstretch, kv3]]]
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*kpos* -- controller 4, pickup position. Value range of kv1 is 0-127.

*ksus* -- controller11, string sustain. Value range of kv2 is 0-127.

*kstretch* -- controller 1, string stretch. Value range of kv3 is 0-127.



### Note

The code for this opcode is taken directly from the StifKarp class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the controller values are krate variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKStifKarp opcode. It uses the file *STKStifKarp.csd* [examples/STKStifKarp.csd].

### Example 941. Example of the STKStifKarp opcode.

```
<CsoundSynthesizer>
```

```
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKStifKarp.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifrq = p4
kv1 line p6, p3, p7 ;Pickup Position
kv2 = p5 ;String Sustain

asig STKStifKarp cspch(p4), 1, 4, kv1, 11, kv2, 1, 10
outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 2 5.00 0 100 100
i 1 + 40 5.00 127 1 127
i 1 10 32 5.00 127 1 10
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Michael Gogins (after Perry Cook)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKTubeBell

STKTubeBell — STKTubeBell is a tubular bell (orchestral chime) FM synthesis instrument.

## Description

STKTubeBell is a tubular bell (orchestral chime) FM synthesis instrument. It uses two simple FM Pairs summed together, also referred to as algorithm 5 of the TX81Z.

## Syntax

```
asignal STKTubeBell ifrequency, iamplitude, [kmod, kv1[, kcross, kv2[, klfo, kv3[, klfodepth, kv4[, ka
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*kmod* -- controller 2, modulator index 1. Value range of kv1 is 0-127.

*kcross* -- controller 4, crossfade of outputs. Value range of kv2 is 0-127.

*klfo* -- controller 11, speed of low-frequency oscillator. Value range of kv3 is 0-127.

*klfodepth* -- controller 1, depth of low-frequency oscillator. Value range of kv4 is 0-127.

*kadsr* -- controller 128, ADSR 2 and 4 target. Value range of kv5 is 0-127.



### Note

The code for this opcode is taken directly from the TubeBell class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the controller values are krate variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKTubeBell opcode. It uses the file *STKTubeBell.csd* [examples/STKTubeBell.csd].

**Example 942. Example of the STKTubeBell opcode.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKTubeBell.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifrq = p4
kv2 line p6, p3, p7 ;Crossfade of Outputs
kv1 = p5 ;(FM) Modulator Index One

asig STKTubeBell cpspch(p4), 1, 2, kv1, 4, kv2, 11, 10, 1, 70, 128,50
outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 2 7.05 0 100 100
i 1 + 4 9.00 127 127 30
i 1 + 1 10.00 127 12 30
i 1 + 3 6.08 127 1 100
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Michael Gogins (after Perry Cook)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKVoicForm

STKVoicForm — STKVoicForm is a four formant synthesis instrument.

## Description

STKVoicForm is a four formant synthesis instrument. This instrument contains an excitation singing wavetable (looping wave with random and periodic vibrato, smoothing on frequency, etc.), excitation noise, and four sweepable complex resonances. Measured formant data is included, and enough data is there to support either parallel or cascade synthesis. In the floating point case cascade synthesis is the most natural so that's what you'll find here.

## Syntax

```
asignal STKVoicForm ifrequency, iamplitude, [kmix, kv1[, ksel, kv2[, klfo, kv3[, klfodepth, kv4[, kloud
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*kmix* -- controller 2, voiced/unvoiced mix. Value range of kv1 is 0-127.

*ksel* -- controller 4, vowel/phoneme selection. Value range of kv2 is 0-127.

*klfo* -- controller 11, speed of low-frequency oscillator. Value range of kv3 is 0-127.

*klfodepth* -- controller 1, depth of low-frequency oscillator. Value range of kv4 is 0-127.

*kloud* -- controller 128, loudness (spectral tilt). Value range of kv5 is 0-127.



### Note

The code for this opcode is taken directly from the VoicForm class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the controller values are krate variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKVoicForm opcode. It uses the file *STKVoicForm.csd* [examples/STKVoicForm.csd].

**Example 943. Example of the STKVoicForm opcode.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKVoicForm.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifrq = p4
kv2 line p5, p3, p6      ;Vowel/Phoneme Selection

asig STKVoicForm cpspch(p4), 1, 2, 1, 4, kv2, 128, 100, 1, 10, 11, 100
asig = asig * .5      ;too loud
outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 5 7.00 100 0
i 1 + 10 7.00 1 50
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Michael Gogins (after Perry Cook)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKWhistle

STKWhistle — STKWhistle produces whistle sounds.

## Description

STKWhistle produces (police) whistle sounds. It uses a hybrid physical/spectral model of a police whistle (a la Cook).

## Syntax

```
asignal STKWhistle ifrequency, iamplitude, [kmod, kv1[, knoise, kv2[, kfipfreq, kv3[, kfipgain, kv4[, k
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*kmod* -- controller 2, blowing frequency modulation. Value range of kv1 is 0-127.

*knoise* -- controller 4, noise gain. Value range of kv2 is 0-127.

*kfipfreq* -- controller 11, fipple modulation frequency. Value range of kv3 is 0-127.

*kfipgain* -- controller 1, fipple modulation gain. Value range of kv4 is 0-127.

*kvol* -- controller 128, volume. Value range of kv5 is 0-127.



### Note

The code for this opcode is taken directly from the Whistle class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the controller values are krate variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKWhistle opcode. It uses the file *STKWhistle.csd* [examples/STKWhistle.csd].

**Example 944. Example of the STKWhistle opcode.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKWhistle.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifrq = p4
kv1 line p5, p3, p6      ;Blowing Frequency Modulation
kv3 = p7                ;Fipple Modulation Frequency

asig STKWhistle cpspch(p4), 1, 4, 20, 11, kv3, 1, 100, 2, kv1, 128, 127
asig = asig*.7         ;too loud
outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 .5 9.00 100 30 30
i 1 1 3 9.00 100 0 20
i 1 4.5 . 9.00 1 0 100
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Michael Gogins (after Perry Cook)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# STKWurley

STKWurley — STKWurley simulates a Wurlitzer electric piano FM synthesis instrument.

## Description

STKWurley simulates a Wurlitzer electric piano FM synthesis instrument. It uses two simple FM Pairs summed together, also referred to as algorithm 5 of the TX81Z.

## Syntax

```
asignal STKWurley ifrequency, iamplitude, [kmod, kv1[, kcross, kv2[, klfo, kv3[, klfodepth, kv4[, kadsr
```

## Initialization

*ifrequency* -- Frequency of note played, in Hertz.

*iamplitude* -- Amplitude of note played (range 0-1).

## Performance

*kmod* -- controller 2, modulator index 1. Value range of kv1 is 0-127.

*kcross* -- controller 4, crossfade of outputs. Value range of kv2 is 0-127.

*klfo* -- controller 11, speed of low-frequency oscillator. Value range of kv3 is 0-127.

*klfodepth* -- controller1, depth of low-frequency oscillator. Value range of kv4 is 0-127.

*kadsr* -- controller 128, ADSR 2 and 4 target. Value range of kv5 is 0-127.



### Note

The code for this opcode is taken directly from the Wurley class in the Synthesis Toolkit in C++ by Perry R. Cook and Gary P. Scavone. More on the STK classes can be found here: <https://ccrma.stanford.edu/software/stk/classes.html>

*kc1, kv1, kc2, kv2, kc3, kv3, kc4, kv4, kc5, kv5, kc6, kv6, kc7, kv7, kc8, kv8* -- Up to 8 optional k-rate controller pairs for the STK opcodes. Each controller pair consists of a controller number (kc) followed by a controller value (kv). Both the controller numbers and the controller values are krate variables. However, during a performance, normally the controller numbers stay fixed while the corresponding controller values may change at any time. The order of the controller pair is arbitrary, as long as they are after *iamplitude*. Also, it is not needed that all controller pairs are used.

## Examples

Here is an example of the STKWurley opcode. It uses the file *STKWurley.csd* [examples/STKWurley.csd].

**Example 945. Example of the STKWurley opcode.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o STKWurley.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifrq = p4
kv1 line p5, p3, p6      ;(FM) Modulator Index One
kv3 = p7

asig STKWurley cpspch(p4), 1, 2,kv1, 4, 10, 11, kv3, 1, 30, 128, 75
outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 .5 7.00 75 0 20
i 1 + . 8.00 120 0 20
i 1 + 1 6.00 50 120 20
i 1 + 4 8.00 10 10 127
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*STKRhodey*.

## Credits

Author: Michael Gogins (after Perry Cook)  
Irreducible Productions  
New York, NY

New in Csound version 5.11

# strchar

strchar — Return the ASCII code of a character in a string

## Description

Return the ASCII code of the character in Sstr at ipos (defaults to zero which means the first character), or zero if ipos is out of range. strchar runs at init time only.

## Syntax

```
ichr strchar Sstr[, ipos]
```

## Examples

Here is an example of the strchar opcode. It uses the file *strchar.csd* [examples/strchar.csd].

### Example 946. Example of the strchar opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-ndm0
</CsOptions>
<CsInstruments>
;example by joachim heintz 2013

opcode ToAscii, S, S ;returns the ASCII numbers of the input string as string

Sin      xin      ;input string
ilen     strlen   Sin      ;its length
ipos     =        0      ;set counter to zero
Sres     =        ""     ;initialize output string
loop:    ;for all characters in input string:
ichr     strchar  Sin, ipos ;get its ascii code number
Snew     sprintf  "%d ", ichr ;put this number into a new string
Sres     strcat  Sres, Snew ;append this to the output string
loop_lt  ipos, 1, ilen, loop ;see comment for 'loop:'
xout     xout    Sres     ;return output string

endop

instr Characters

printf_i "\nCharacters:\n given as single strings: %s%s%s%s%s\n", 1, "c", "s", "o", "u", "n", "d"
printf_i " but can also be given as numbers: %c%c%c%c%c\n", 1, 99, 115, 111, 117, 110, 100
Scsound ToAscii "csound"
printf_i " in csound, the ASCII code of a character can be accessed with the opcode strchar.%s", 1, "c"
printf_i " the name 'csound' returns the numbers %s\n\n", 1, Scsound
endin

</CsInstruments>
<CsScore>

i "Characters" 0 0
e
```

```
</CsScore>  
</CsoundSynthesizer>
```

## See also

*strchark*

## Credits

Author: Istvan Varga  
2006

# strchark

strchark — Return the ASCII code of a character in a string

## Description

Return the ASCII code of the character in Sstr at kpos (defaults to zero which means the first character), or zero if kpos is out of range. strchark runs both at init and performance time.

## Syntax

```
kchr strchark Sstr[, kpos]
```

## See also

*strchar*

## Credits

Author: Istvan Varga  
2006

New in version 5.02

# strcpy

strcpy — Assign value to a string variable

## Description

Assign to a string variable by copying the source which may be a constant or another string variable. strcpy and = copy the string at i-time only.

## Syntax

```
Sdst strcpy Ssrc
```

```
Sdst = Ssrc
```

## Example

```
Sfoo  strcpy "Hello, world !"  
      puts  Sfoo, 1
```

## See also

*strcpyk*

## Credits

Author: Istvan Varga  
2005

# strcpyk

strcpyk — Assign value to a string variable (k-rate)

## Description

Assign to a string variable by copying the source which may be a constant or another string variable. *strcpyk* does the assignment both at initialization and performance time.

## Syntax

```
Sdst strcpyk Ssrc
```

## Examples

Here is an example of the *strcpyk* opcode. It uses the file *strcpyk.csd* [examples/strcpyk.csd].

### Example 947. Example of the *strcpyk* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o strcpyk.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

seed 0

instr 1
;get one element of the input string whenever the metro
;triggers, and call a subinstrument to play the file

Smember   strget   p4
istrlen   strlen   Smember
kprint    init     0
ktrig     metro    .6

;whenever the trigger gives signal
if ktrig == 1 then
;choose a random element (0, 1 or 2)
kel       random   0, 3.9999
kel       =        int(kel)
;make a copy for leaving Smember intact
Scopy    strcpyk   Smember
;set the initial index for reading substrings
kndx     =         0
```

```

;set counter for searching the element
kcount = 0
;start looping over the elements in Smember
loop:
kdelim strindexk Scopy, ":"
;as long as ":" occurs in Scopy, do:
if kdelim > 0 then
;if this is the element to get
if kel == kcount then
;read it as substring
Sfile strsubk Scopy, kndx, kdelim
kprint = kprint+1
;and jump out
kgoto call
;if not
else
;cut off this element from Scopy
Scopy strsubk Scopy, kdelim+1, istrlen
endif
;if no element has been found,go back to loop
;and look for the next element
kcount = kcount+1
kgoto loop
;if there is no delimiter left, the rest is the searched element
else
Sfile strcpyk Scopy
endif
call:
;print the result, call the subinstrument and play the file
printf "kel = %d, file = '%s'\n", ktrig+kprint, kel, Sfile
S_call sprintfk {{i 2 0 1 "%s"}}, Sfile
scoreline S_call, ktrig
endif
endin

instr 2 ;play
Sfile strget p4
ilen filelen Sfile
p3 = ilen
asig soundin Sfile
outs asig, asig
endin
</CsInstruments>
<CsScore>

i 1 0 30 "mary.wav:fox.wav:beats.wav:flute.aiff"
e
</CsScore>
</CsoundSynthesizer>

```

## See also

*strcpy*

## Credits

Author: Istvan Varga  
2005

# strcat

strcat — Concatenate strings

## Description

Concatenate two strings and store the result in a variable. *strcat* runs at i-time only. It is allowed for any of the input arguments to be the same as the output variable.

## Syntax

```
Sdst strcat Ssrc1, Ssrc2
```

## Examples

Here is an example of the *strcat* opcode. It uses the file *strcat.csd* [examples/strcat.csd].

### Example 948. Example of the *strcat* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o strcat.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

Sname = "beats"
Sname strcat Sname, ".wav"
asig soundin Sname
      outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 2
e
</CsScore>
</CsoundSynthesizer>
```

## See also

*strcatk*

## Credits

Author: Istvan Varga  
2005

New in version 5.02

# strcatk

strcatk — Concatenate strings (k-rate)

## Description

Concatenate two strings and store the result in a variable. *strcatk* does the concatenation both at initialization and performance time. It is allowed for any of the input arguments to be the same as the output variable.

## Syntax

```
Sdst strcatk Ssrc1, Ssrc2
```

## Examples

Here is an example of the *strcatk* opcode. It uses the file *strcatk.csd* [examples/strcatk.csd].

### Example 949. Example of the *strcatk* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o strcatk.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

  sr      = 48000
  ksmps   = 16
  nchnls  = 2
  Odbfs   = 1

; Example by Jonathan Murphy 2007

  instr 1

  S1      = "1"
  S2      = " + 1"
  ktrig   init      0
  kval    init      2
  if (ktrig == 1) then
    S1    strcatk   S1, S2
    kval  = kval + 1
  endif
  String  sprintfk  "%s = %d", S1, kval
  puts    String, kval
  ktrig   metro    1

  endin

</CsInstruments>
<CsScore>
```

```
i1 0 10  
e  
</CsScore>  
</CsoundSynthesizer>
```

Its output should include lines like this:

```
1 + 1 = 2  
1 + 1 + 1 = 3  
1 + 1 + 1 + 1 = 4  
1 + 1 + 1 + 1 + 1 = 5  
1 + 1 + 1 + 1 + 1 + 1 = 6  
1 + 1 + 1 + 1 + 1 + 1 + 1 = 7  
1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 8  
1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 9  
1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 10  
1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 11  
1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 12
```

## See also

*strcat*

## Credits

Author: Istvan Varga  
2005

New in version 5.02

# strcmp

strcmp — Compare strings

## Description

Compare strings and set the result to -1, 0, or 1 if the first string is less than, equal to, or greater than the second, respectively. strcmp compares at i-time only.

## Syntax

```
ires strcmp S1, S2
```

## Examples

Here is an example of the strcmp opcode. It uses the file *strcmp.csd* [examples/strcmp.csd].

### Example 950. Example of the strcmp opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o strcmp.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
;modified example from Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

opcode Triad, iii, S ;define UDO
Sname      xin
iMaj       strcmp   "maj", Sname
iMin       strcmp   "min", Sname
iPrim      =        8.00 ;notes in pitch notation
iQuint     =        8.05
  if iMaj == 0 then
iTer       =        8.03
  elseif iMin == 0 then
iTer       =        8.02
  endif
  xout      iPrim, iTer, iQuint
endop

instr 1

Sname strget p4
ia, ib, ic Triad Sname ;apply UDO
print ia, ib, ic
asig1 pluck 0.7, cpspch(ia), 220, 0, 1
asig2 pluck 0.7, cpspch(ib), 220, 0, 1
asig3 pluck 0.7, cpspch(ic), 220, 0, 1
asig = (asig1+asig2+asig3)*.5
```

```
    outs asig, asig

endin
</CsInstruments>
<CsScore>
i1 0 3 "maj"
i1 4 3 "min"
</CsScore>
</CsoundSynthesizer>
```

## See also

*strecmpk*

## Credits

Author: Istvan Varga  
2005

# strcmpk

strcmp — Compare strings

## Description

Compare strings and set the result to -1, 0, or 1 if the first string is less than, equal to, or greater than the second, respectively. *strcmpk* does the comparison both at initialization and performance time.

## Syntax

```
kres strcmpk S1, S2
```

## See also

*strcmp*

## Credits

Author: Istvan Varga  
2005

# streson

streson — A string resonator with variable fundamental frequency.

## Description

An audio signal is modified by a string resonator with variable fundamental frequency.

## Syntax

```
%  
ares streson asig, kfr, kfdbgain
```

## Performance

*asig* -- the input audio signal.

*kfr* -- the fundamental frequency of the string.

*kfdbgain* -- feedback gain, typically between 0 and 1, of the internal delay line. A value close to 1 creates a slower decay and a more pronounced resonance. Small values may leave the input signal unaffected. Depending on the filter frequency, typical values are > 0.9. Values down to -1 are also useful.

*streson* passes the input *asig* through a network composed of comb, low-pass and all-pass filters, similar to the one used in some versions of the Karplus-Strong algorithm, creating a string resonator effect. The fundamental frequency of the “string” is controlled by the k-rate variable *kfr*. This opcode can be used to simulate sympathetic resonances to an input signal.

See *Modal Frequency Ratios* for frequency ratios of real instruments which can be used to determine the values of *kfrq*.

*streson* is an adaptation of the StringFlt object of the SndObj Sound Object Library developed by the author.

## Examples

Here is an example of the *streson* opcode. It uses the file *streson.csd* [examples/streson.csd].

### Example 951. Example of the *streson* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
-odac ;;realtime audio out  
;-iadc ;;uncomment -iadc if realtime audio input is needed too  
; For Non-realtime ouput leave only the line below:  
; -o streson.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>
```

```
sr = 44100
ksmps = 32
Odbfs = 1
nchnls = 2

instr 1

asig diskin2 "fox.wav", 1, 0, 1

kfr = p4
ifdbgain = 0.90

astr streson asig, kfr, ifdbgain
asig clip astr, 0, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 1 20
i 1 + . >
i 1 + . >
i 1 + . >
i 1 + . >
i 1 + . >
i 1 + . >
i 1 + . 1000
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
Music Department  
National University of Ireland, Maynooth  
Maynooth, Co. Kildare  
1998

New in Csound version 3.494

# strfromurl

strfromurl — Set string variable to value read from an URL

## Description

*strfromurl* sets a string variable at initialization time to the value found from reading an URL.

## Syntax

```
Sdst strfromurl StringURL
```

## Initialization

*StringURL* -- string naming an URL.

*Sdst* -- destination string variable

## Examples

Here is an example of the strfromurl opcode. It uses the file *strfromurl.csd* [examples/strfromurl.csd].

### Example 952. Example of the strfromurl opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o strget.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

#include "http://codemist.co.uk/jpff/test.in"

instr 2

Sfile strfromurl "http://codemist.co.uk/jpff/test.in"
  prints Sfile
endin
</CsInstruments>
<CsScore>

i 1 0 1
i 2 1 1
e
</CsScore>
```

`</CsoundSynthesizer>`

## See also

*strset*

## Credits

Author: John ffitch

2013

URLs available from Csound 6.02 if built with support

# strget

strget — Set string variable to value from strset table or string p-field

## Description

*strget* sets a string variable at initialization time to the value stored in *strset* table at the specified index, or a string p-field from the score. If there is no string defined for the index, the variable is set to an empty string.

## Syntax

```
Sdst strget indx
```

## Initialization

*indx* -- strset index, or score p-field

*Sdst* -- destination string variable

## Examples

Here is an example of the strget opcode. It uses the file *strget.csd* [examples/strget.csd].

### Example 953. Example of the strget opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o strget.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

strset 1, "fox.wav"
strset 2, "beats.wav"

instr 1

Sfile strget p4
asig soundin Sfile
outs asig, asig

endin
</CsInstruments>
<CsScore>
```

```
i 1 0 2.7 1
i 1 + 2 2
e
</CsScore>
</CsoundSynthesizer>
```

## See also

*strset*

## Credits

Author: Istvan Varga

2005

# strindex

strindex — Return the position of the first occurrence of a string in another string

## Description

Return the position of the first occurrence of S2 in S1, or -1 if not found. If S2 is empty, 0 is returned. strindex runs at init time only.

## Syntax

```
ipos strindex S1, S2
```

## See also

*strindexk*

## Credits

Author: Istvan Varga  
2006

New in version 5.02

# strindexk

strindexk — Return the position of the first occurrence of a string in another string

## Description

Return the position of the first occurrence of S2 in S1, or -1 if not found. If S2 is empty, 0 is returned. strindexk runs both at init and performance time.

## Syntax

```
kpos strindexk S1, S2
```

## Examples

Here is an example of the strindexk opcode. It uses the file *strindexk.csd* [examples/strindexk.csd].

### Example 954. Example of the strindexk opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o strindexk.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

seed 0

instr 1
;get one element of the input string whenever the metro
;triggers, and call a subinstrument to play the file

Smember   strget   p4
istrlen   strlen   Smember
kprint    init    0
ktrig     metro   .5

;whenever the trigger gives signal
if ktrig == 1 then
;choose a random element (0, 1 or 2)
kel       random   0, 2.9999
kel       =         int(kel)
;make a copy for leaving Smember intact
Scopy     strcpyk  Smember
;set the initial index for reading substrings
kndx     =         0
```

```

;set counter for searching the element
kcount = 0
;start looping over the elements in Smember
loop:
kdelim strindexk Scopy, ":"
;as long as ":" occurs in Scopy, do:
if kdelim > 0 then
;if this is the element to get
if kel == kcount then
;read it as substring
Sfile strsubk Scopy, kndx, kdelim
kprint = kprint+1
;and jump out
kgoto call
;if not
else
;cut off this element from Scopy
Scopy strsubk Scopy, kdelim+1, istrlen
endif
;if no element has been found,go back to loop
;and look for the next element
kcount = kcount+1
kgoto loop
;if there is no delimiter left, the rest is the searched element
else
Sfile strcpyk Scopy
endif
call:
;print the result, call the subinstrument and play the file
printf "kel = %d, file = '%s'\n", ktrig+kprint, kel, Sfile
S_call sprintfk {{i 2 0 1 "%s"}}, Sfile
scoreline S_call, ktrig
endif
endin

instr 2 ;play
Sfile strget p4
ilen filelen Sfile
p3 = ilen
asig soundin Sfile
outs asig, asig
endin
</CsInstruments>
<CsScore>

i 1 0 30 "mary.wav:fox.wav:beats.wav"
e
</CsScore>
</CsoundSynthesizer>

```

## See also

*strindex*

## Credits

Author: Istvan Varga  
2006

New in version 5.02

# strlen

strlen — Return the length of a string

## Description

Return the length of a string, or zero if it is empty. strlen runs at init time only.

## Syntax

```
ilen strlen Sstr
```

## See also

*strlenk*

## Credits

Author: Istvan Varga  
2006

New in version 5.02

# strlen

strlen — Return the length of a string

## Description

Return the length of a string, or zero if it is empty. strlen runs both at init and performance time.

## Syntax

```
klen strlen Sstr
```

## See also

*strlen*

## Credits

Author: Istvan Varga  
2006

New in version 5.02

# strlower

strlower — Convert a string to lower case

## Description

Convert Ssrc to lower case, and write the result to Sdst. strlower runs at init time only.

## Syntax

```
Sdst strlower Ssrc
```

## Examples

Here is an example of the strlower opcode. It uses the file *strlower.csd* [examples/strlower.csd].

### Example 955. Example of the strlower opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n      ;;no sound output
;-iadc  ;;uncomment -iadc if realtime audio input is needed too
</CsOptions>
<CsInstruments>
;example of Joachim Heintz

    opcode FilSuf, S, So
    ;returns the suffix of a filename or path, optional in lower case
    Spath, ilow xin
    ipos strindex Spath, "." ;look for the rightmost '.'
    Suf strsub Spath, ipos+1 ;extract the substring after "."
    if ilow != 0 then ;if ilow input is not 0 then
    Suf strlower Suf ;convert to lower case
    endif
    xout Suf
    endop

instr suff

ilow = p4
prints "Printing suffix:\n"
Suf FilSuf "/my/dir/my/file.WAV", ilow
puts Suf, 1

endin
</CsInstruments>
<CsScore>
i "suff" 0 1 0 ;do not convert to lower case
i "suff" 3 1 1 ;convert to lower case
e
</CsScore>
</CsoundSynthesizer>
```

## See also

*strlowerk*

## Credits

Author: Istvan Varga  
2006

New in version 5.02

# strlowerk

strlowerk — Convert a string to lower case

## Description

Convert Ssrc to lower case, and write the result to Sdst. strlowerk runs both at init and performance time.

## Syntax

```
Sdst strlowerk Ssrc
```

## See also

*strlower*

## Credits

Author: Istvan Varga  
2006

New in version 5.02

# strrindex

strrindex — Return the position of the last occurrence of a string in another string

## Description

Return the position of the last occurrence of S2 in S1, or -1 if not found. If S2 is empty, the length of S1 is returned. strrindex runs at init time only.

## Syntax

```
ipos strrindex S1, S2
```

## Examples

Here is an example of the strrindex opcode. It uses the file *strrindex.csd* [examples/strrindex.csd].

### Example 956. Example of the strrindex opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n      ;;no sound output
;-iadc  ;;uncomment -iadc if realtime audio input is needed too
</CsOptions>
<CsInstruments>
;example by Joachim Heintz

    opcode FilNam, S, S
;returns the name of a file path
Spath xin
ipos  strrindex Spath, "/" ;look for the rightmost '/'
Snam  strsub  Spath, ipos+1 ;extract the substring
xout      Snam
    endop

instr name
prints      "Printing name:\n"
Snam FilNam  "/my/dir/my/file.WAV"
puts       Snam, 1

endin
</CsInstruments>
<CsScore>
i "name" 0 0
e
</CsScore>
</CsoundSynthesizer>
```

## See also

*strrindexk*

## Credits

Author: Istvan Varga  
2006

New in version 5.02

# strrindexk

strrindexk — Return the position of the last occurrence of a string in another string

## Description

Return the position of the last occurrence of S2 in S1, or -1 if not found. If S2 is empty, the length of S1 is returned. strrindexk runs both at init and performance time.

## Syntax

```
kpos strrindexk S1, S2
```

## See also

*strrindex*

## Credits

Author: Istvan Varga  
2006

New in version 5.02

# strset

strset — Allows a string to be linked with a numeric value.

## Description

Allows a string to be linked with a numeric value.

## Syntax

```
strset iarg, istring
```

## Initialization

*iarg* -- the numeric value.

*istring* -- the alphanumeric string (in double-quotes).

*strset* (optional) allows a string, such as a filename, to be linked with a numeric value. Its use is optional.

## Examples

The following statement, used in the orchestra header, will allow the numeric value 10 to be substituted anywhere the soundfile *asound.wav* is called for.

```
strset 10, "asound.wav"
```

## Examples

Here is an example of the *strset* opcode. It uses the file *strset.csd* [examples/strset.csd].

### Example 957. Example of the *strset* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d          ;;RT audio I/O
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 1
nchnls = 1

;Example by Andres Cabrera 2008

; \\n is used to denote "new line"
strset 1, "String 1\\n"
```

```
strset 2, "String 2\\n"

instr 1
Str strget p4
prints Str
endin

</CsInstruments>
<CsScore>
;      p4 is used to select string
i 1 0 1 1
i 1 3 1 2
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pset* and *strget*

# strsub

strsub — Extract a substring

## Description

Return a substring of the source string. strsub runs at init time only.

## Syntax

```
Sdst strsub Ssrc[, istart[, iend]]
```

## Initialization

*istart* (optional, defaults to 0) -- start position in Ssrc, counting from 0. A negative value means the end of the string.

*iend* (optional, defaults to -1) -- end position in Ssrc, counting from 0. A negative value means the end of the string. If iend is less than istart, the output is reversed.

## Examples

Here is an example of the strsub opcode. It uses the file *strsub.csd* [examples/strsub.csd].

### Example 958. Example of the strsub opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc      ;;-d      RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o strsub.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
; By: Jonathan Murphy 2007

instr 1
  Smember strget p4

  ; Parse Smember
  istrlen  strlen  Smember
  idelimiter strindex Smember, ":"

  S1      strsub Smember, 0, idelimiter ; "String1"
  S2      strsub Smember, idelimiter + 1, istrlen ; "String2"

  printf "First string: %s\nSecond string: %s\n", 1, S1, S2

endin

</CsInstruments>
```

```
<CsScore>  
i 1 0 1 "String1:String2"  
</CsScore>  
</CsoundSynthesizer>
```

## See also

*strsubk*

## Credits

Author: Istvan Varga  
2006

# strsubk

strsubk — Extract a substring

## Description

Return a substring of the source string. strsubk runs both at init and performance time.

## Syntax

```
Sdst strsubk Ssrc, kstart, kend
```

## Performance

*kstart* -- start position in Ssrc, counting from 0. A negative value means the end of the string.

*kend* -- end position in Ssrc, counting from 0. A negative value means the end of the string. If *kend* is less than *kstart*, the output is reversed.

## See also

*strsub*

## Credits

Author: Istvan Varga  
2006

# strtod

strtod — Converts a string to a float (i-rate).

## Description

Convert a string to a floating point value. It is also possible to pass an strset index or a string p-field from the score instead of a string argument. If the string cannot be parsed as a floating point or integer number, an init or perf error occurs and the instrument is deactivated.

## Syntax

```
ir strtod Sstr
```

```
ir strtod indx
```

## Initialization

*Sstr* -- String to convert.

*indx* -- index of string set by strset

## Performance

*ir* -- Value of string as float.

## Credits

Author: Istvan Varga  
2005

# strtodk

strtodk — Converts a string to a float (k-rate).

## Description

Convert a string to a floating point value at i- or k-rate. It is also possible to pass an strset index or a string p-field from the score instead of a string argument. If the string cannot be parsed as a floating point or integer number, an init or perf error occurs and the instrument is deactivated.



### Note

If a k-rate index variable is used, it should be valid at i-time as well.

## Syntax

```
kr strtodk Sstr
```

```
kr strtodk kndx
```

## Performance

*kr* -- Value of string as float.

*Sstr* -- String to convert.

*indx* -- index of string set by strset

## Credits

Author: Istvan Varga  
2005

# strtol

strtol — Converts a string to a signed integer (i-rate).

## Description

Convert a string to a signed integer value. It is also possible to pass an strset index or a string p-field from the score instead of a string argument. If the string cannot be parsed as an integer number, an init error occurs and the instrument is deactivated.

## Syntax

```
ir strtol Sstr
```

```
ir strtol indx
```

## Initialization

*Sstr* -- String to convert.

*indx* -- index of string set by strset

strtol can parse numbers in decimal, octal (prefixed by 0), and hexadecimal (with a prefix of 0x) format.

## Performance

*ir* -- Value of string as signed integer.

## Credits

Author: Istvan Varga  
2005

# strtolk

strtolk — Converts a string to a signed integer (k-rate).

## Description

Convert a string to a signed integer value at i- or k-rate. It is also possible to pass an strset index or a string p-field from the score instead of a string argument. If the string cannot be parsed as an integer number, an init or perf error occurs and the instrument is deactivated.



### Note

If a k-rate index variable is used, it should be valid at i-time as well.

## Syntax

```
kr strtolk Sstr
```

```
kr strtolk kndx
```

strtolk can parse numbers in decimal, octal (prefixed by 0), and hexadecimal (with a prefix of 0x) format.

## Performance

*kr* -- Value of string as signed integer.

*Sstr* -- String to convert.

*indx* -- index of string set by strset

## Credits

Author: Istvan Varga  
2005

# strupper

strupper — Convert a string to upper case

## Description

Convert Ssrc to upper case, and write the result to Sdst. strupper runs at init time only.

## Syntax

```
Sdst strupper Ssrc
```

## See also

*strupperk*

## Credits

Author: Istvan Varga  
2006

New in version 5.02

# strupperk

strupperk — Convert a string to upper case

## Description

Convert Ssrc to upper case, and write the result to Sdst. strupperk runs both at init and performance time.

## Syntax

```
Sdst strupperk Ssrc
```

## See also

*strupper*

## Credits

Author: Istvan Varga  
2006

New in version 5.02

# subinstr

subinstr — Creates and runs a numbered instrument instance.

## Description

Creates an instance of another instrument and is used as if it were an opcode.

## Syntax

```
a1, [...] [, a8] subinstr instrnum [, p4] [, p5] [...]  
a1, [...] [, a8] subinstr "insname" [, p4] [, p5] [...]
```

## Initialization

*instrnum* -- Number of the instrument to be called.

*"insname"* -- A string (in double-quotes) representing a named instrument.

## Performance

*a1, ..., a8* -- The audio output from the called instrument. This is generated using the *signal output* opcodes.

*p4, p5, ...* -- Additional input values the are mapped to the called instrument p-fields, starting with p4.

The called instrument's p2 and p3 values will be identical to the host instrument's values. While the host instrument can *control its own duration*, any such attempts inside the called instrument will most likely have no effect.

## See Also

*event, schedule, subinstrinit*

## Examples

Here is an example of the subinstr opcode. It uses the file *subinstr.csd* [examples/subinstr.csd].

### Example 959. Example of the subinstr opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
; Audio out  Audio in  
-odac          -iadc      ;;RT audio I/O  
; For Non-realtime ouput leave only the line below:  
; -o subinstr.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>
```

```

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - Creates a basic tone.
instr 1
; Print the value of p4, should be equal to
; Instrument #2's iamp field.
print p4

; Print the value of p5, should be equal to
; Instrument #2's ipitch field.
print p5

; Create a tone.
asig oscils p4, p5, 0

out asig
endin

; Instrument #2 - Demonstrates the subinstr opcode.
instr 2
iamp = 20000
ipitch = 440

; Use Instrument #1 to create a basic sine-wave tone.
; Its p4 parameter will be set using the iamp variable.
; Its p5 parameter will be set using the ipitch variable.
abasic subinstr 1, iamp, ipitch

; Output the basic tone that we have created.
out abasic
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #2 for one second.
i 2 0 1
e

</CsScore>
</CsSoundSynthesizer>

```

Here is an example of the subinstr opcode using a named instrument. It uses the file *subinstr\_named.csd* [examples/subinstr\_named.csd].

### Example 960. Example of the subinstr opcode using a named instrument.

```

<CsSoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out Audio in
-odac -iadc ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o subinstr_named.wav -W ;; for file output any platform

```

```
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument "basic_tone" - Creates a basic tone.
instr basic_tone
; Print the value of p4, should be equal to
; Instrument #2's iamp field.
print p4

; Print the value of p5, should be equal to
; Instrument #2's ipitch field.
print p5

; Create a tone.
asig oscils p4, p5, 0

out asig
endin

; Instrument #1 - Demonstrates the subinstr opcode.
instr 1
iamp = 20000
ipitch = 440

; Use the "basic_tone" named instrument to create a
; basic sine-wave tone.
; Its p4 parameter will be set using the iamp variable.
; Its p5 parameter will be set using the ipitch variable.
abasic subinstr "basic_tone", iamp, ipitch

; Output the basic tone that we have created.
out abasic
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

New in version 4.21

# subinstrinit

subinstrinit — Creates and runs a numbered instrument instance at init-time.

## Description

Same as *subinstr*, but init-time only and has no output arguments.

## Syntax

```
subinstrinit instrnum [, p4] [, p5] [...]
```

```
subinstrinit "insname" [, p4] [, p5] [...]
```

## Initialization

*instrnum* -- Number of the instrument to be called.

*"insname"* -- A string (in double-quotes) representing a named instrument.

*p4, p5, ...* -- Additional input values the are mapped to the called instrument p-fields, starting with p4.

The called instrument's p2 and p3 values will be identical to the host instrument's values. While the host instrument can *control its own duration*, any such attempts inside the called instrument will most likely have no effect.

## See Also

*event, schedule, subinstr*

## Credits

New in version 4.23

# sum

sum — Sums any number of a-rate signals, or array elements.

## Description

Sums any number of a-rate signals, or array elements.

## Syntax

```
ares sum asig1 [, asig2] [, asig3] [...]  
kres sum karr  
ires sum iarr
```

## Performance

*asig1, asig2, ...* -- a-rate signals to be summed (mixed or added).

*karr, iarr* -- Csound 1-dimensional arrays

## Examples

Here is an example of the sum opcode. It uses the file *sum.csd* [examples/sum.csd].

### Example 961. Example of the sum opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
-odac    ;;realtime audio out  
;-iadc   ;;uncomment -iadc if realtime audio input is needed too  
; For Non-realtime ouput leave only the line below:  
; -o sum.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>  
  
sr = 44100  
ksmps = 32  
nchnls = 2  
0dbfs = 1  
  
gisine ftgen 0, 0, 2^10, 10, 1  
  
instr 1  
  
a1  oscili 1, 10.0, gisine ;combine 3 sinusses  
a2  oscili 1, 1.0, gisine ;at different rates  
a3  oscili 1, 3.0, gisine  
ares sum a1, a2, a3 ;sum them  
  
ares = ares*100 ;scale result and
```

```
asig poscil .5, ares+110, gisine ;add to frequency
    outs asig, asig

endin
</CsInstruments>
<CsScore>

i1 0 5
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Gabriel Maldonado  
Italy  
April 1999

New in Csound version 3.54

Array versions new in version 6.09

# sumarray

sumarray — returns the sum of the elements in an array.

## Description

The *sumarray* opcode returns the sum of all elements in a k-rate array array.

## Syntax

```
ksum sumarray karray
```

## Performance

*ksum* -- variable for result.

*karray* -- array for reading.

## Examples

Here is an example of the sumarray opcode. It uses the file *sumarray.csd* [examples/sumarray.csd].

### Example 962. Example of the sumarray opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n
</CsOptions>
<CsInstruments>
;example by joachim heintz

      seed          0

instr 1
;create an array with 10 elements
kArr[] init      10
;fill in random numbers and print them out
kIndx  =          0
  until kIndx == 10 do
kNum   random    0, 10
kArr[kIndx] =      kNum
printf         "kArr[%d] = %10f\n", kIndx+1, kIndx, kNum
kIndx  +=         1
  od
;calculate sum of all values and print it out
kSum   sumarray  kArr
printf         "Sum of all values in kArr = %f\n", kIndx+1, kSum
turnoff

endin
</CsInstruments>
<CsScore>
i1 0 0.1
```

```
e  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*minarray*, *sumarray*, *scalearray*,

## Credits

Author: John ffitth  
October 2011

New in Csound version 5.14

Revised in Csound version 6.00 to work on multidimensional arrays

# svfilter

`svfilter` — A resonant second order filter, with simultaneous lowpass, highpass and bandpass outputs.

## Description

Implementation of a resonant second order filter, with simultaneous lowpass, highpass and bandpass outputs.

## Syntax

```
alow, ahigh, aband svfilter asig, kcf, kq [, iscl] [, iskip]
```

## Initialization

`iscl` -- coded scaling factor, similar to that in *reson*. A non-zero value signifies a peak response factor of 1, i.e. all frequencies other than `kcf` are attenuated in accordance with the (normalized) response curve. A zero value signifies no scaling of the signal, leaving that to some later adjustment (see *balance*). The default value is 0.

`iskip` (optional, default=0) -- initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*svfilter* is a second order state-variable filter, with k-rate controls for cutoff frequency and Q. As Q is increased, a resonant peak forms around the cutoff frequency. *svfilter* has simultaneous lowpass, highpass, and bandpass filter outputs; by mixing the outputs together, a variety of frequency responses can be generated. The state-variable filter, or "multimode" filter was a common feature in early analog synthesizers, due to the wide variety of sounds available from the interaction between cutoff, resonance, and output mix ratios. *svfilter* is well suited to the emulation of "analog" sounds, as well as other applications where resonant filters are called for.

`kcf` -- Cutoff or resonant frequency of the filter, measured in Hz.

`kq` -- Q of the filter, which is defined (for bandpass filters) as bandwidth/cutoff. `kq` should be in a range between 1 and 500. As `kq` is increased, the resonance of the filter increases, which corresponds to an increase in the magnitude and "sharpness" of the resonant peak. When using *svfilter* without any scaling of the signal (where `iscl` is either absent or 0), the volume of the resonant peak increases as Q increases. For high values of Q, it is recommended that `iscl` be set to a non-zero value, or that an external scaling function such as *balance* is used.

*svfilter* is based upon an algorithm in Hal Chamberlin's *Musical Applications of Microprocessors* (Hayden Books, 1985).

## Examples

Here is an example of the `svfilter` opcode. It uses the file *svfilter.csd* [examples/svfilter.csd].

**Example 963. Example of the svfilter opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o svfilter.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Orchestra file for resonant filter sweep of a sawtooth-like waveform.
; The seperate outputs of the filter are scaled by values from the score,
; and are mixed together.
sr = 44100
kr = 2205
ksmps = 20
nchnls = 1

instr 1

  idur      = p3
  ifreq     = p4
  iamp      = p5
  ilowamp   = p6           ; determines amount of lowpass output in signal
  ihighamp  = p7           ; determines amount of highpass output in signal
  ibandamp  = p8           ; determines amount of bandpass output in signal
  iq        = p9           ; value of q

  iharms    =      (sr*.4) / ifreq

  asig      gbuzz 1, ifreq, iharms, 1, .9, 1           ; Sawtooth-like waveform
  kfreq     linseg 1, idur * 0.5, 4000, idur * 0.5, 1 ; Envelope to control filter cutoff

  alow, ahigh, aband  svfilter asig, kfreq, iq

  aout1     =      alow * ilowamp
  aout2     =      ahigh * ihighamp
  aout3     =      aband * ibandamp
  asum      =      aout1 + aout2 + aout3
  kenv      linseg 0, .1, iamp, idur -.2, iamp, .1, 0 ; Simple amplitude envelope
  out      asum * kenv

endin

</CsInstruments>
<CsScore>

f1 0 8192 9 1 1 .25

i1 0 5 100 1000 1 0 0 5 ; lowpass sweep
i1 5 5 200 1000 1 0 0 30 ; lowpass sweep, octave higher, higher q
i1 10 5 100 1000 0 1 0 5 ; highpass sweep
i1 15 5 200 1000 0 1 0 30 ; highpass sweep, octave higher, higher q
i1 20 5 100 1000 0 0 1 5 ; bandpass sweep
i1 25 5 200 1000 0 0 1 30 ; bandpass sweep, octave higher, higher q
i1 30 5 200 2000 .4 .6 0 ; notch sweep - notch formed by combining highpass and lowpass outputs
e

</CsScore>

```

</CsoundSynthesizer>

## Credits

Author: Sean Costello  
Seattle, Washington  
1999

New in Csound version 3.55

iskip new in version 6.06

# syncgrain

syncgrain — Synchronous granular synthesis.

## Description

*syncgrain* implements synchronous granular synthesis. The source sound for the grains is obtained by reading a function table containing the samples of the source waveform. For sampled-sound sources, *GENO1* is used. *syncgrain* will accept deferred allocation tables.

The grain generator has full control of frequency (grains/sec), overall amplitude, grain pitch (a sampling increment) and grain size (in secs), both as fixed or time-varying (signal) parameters. An extra parameter is the grain pointer speed (or rate), which controls which position the generator will start reading samples in the table for each successive grain. It is measured in fractions of grain size, so a value of 1 (the default) will make each successive grain read from where the previous grain should finish. A value of 0.5 will make the next grain start at the midway position from the previous grain start and finish, etc.. A value of 0 will make the generator read always from a fixed position of the table (wherever the pointer was last at). A negative value will decrement pointer positions. This control gives extra flexibility for creating timescale modifications in the resynthesis.

*syncgrain* will generate any number of parallel grain streams (which will depend on grain density/frequency), up to the *iolaps* value (default 100). The number of streams (overlapped grains) is determined by  $\text{grainsize} * \text{grain\_freq}$ . More grain overlaps will demand more calculations and the synthesis might not run in realtime (depending on processor power).

*syncgrain* can simulate FOF-like formant synthesis, provided that a suitable shape is used as grain envelope and a sinewave as the grain wave. For this use, grain sizes of around 0.04 secs can be used. The formant centre frequency is determined by the grain pitch. Since this is a sampling increment, in order to use a frequency in Hz, that value has to be scaled by  $\text{tablesize}/\text{sr}$ . Grain frequency will determine the fundamental.

*syncgrain* uses floating-point indexing, so its precision is not affected by large-size tables. This opcode is based on the SndObj library SyncGrain class.

## Syntax

```
asig syncgrain kamp, kfreq, kpitch, kgrsize, kprate, ifun1, \  
      ifun2, iolaps
```

## Initialization

*ifun1* -- source signal function table. Deferred-allocation tables (see *GENO1*) are accepted, but the opcode expects a mono source.

*ifun2* -- grain envelope function table.

*iolaps* -- maximum number of overlaps,  $\max(kfreq) * \max(kgrsize)$ . Estimating a large value should not affect performance, but exceeding this value will probably have disastrous consequences.

## Performance

*kamp* -- amplitude scaling

*kfreq* -- frequency of grain generation, or density, in grains/sec.

*kpitch* -- grain pitch scaling (1=normal pitch, < 1 lower, > 1 higher; negative, backwards)

*kgrsize* -- grain size in secs.

*kprate* -- readout pointer rate, in grains. The value of 1 will advance the reading pointer 1 grain ahead in the source table. Larger values will time-compress and smaller values will time-expand the source signal. Negative values will cause the pointer to run backwards and zero will freeze it.

## Examples

Here is an example of the *syncgrain* opcode. It uses the file *syncgrain.csd* [examples/syncgrain.csd].

### Example 964. Example of the *syncgrain* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o syncgrain.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 2

instr 1

iolaps = 2
igrsize = 0.04
ifreq = iolaps/igrsize
ips = 1/iolaps

istr = .3 /* timescale */
ipitch = p4 /* pitchscale */

asig syncgrain 1, ifreq, ipitch, igrsize, ips*istr, 1, 2, iolaps
outs asig, asig

endin
</CsInstruments>
<CsScore>
f1 0 0 1 "fox.wav" 0 0 0 ;deferred table
f2 0 8192 20 2 1

i1 0 5 1
i1 + 5 4
i1 + 5 .8
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini

January 2005

New plugin in version 5

January 2005.

# syncloop

syncloop — Synchronous granular synthesis.

## Description

*syncloop* is a variation on *syncgrain*, which implements synchronous granular synthesis. *syncloop* adds loop start and end points and an optional start position. Loop start and end control grain start positions, so the actual grains can go beyond the loop points (if the loop points are not at the extremes of the table), enabling seamless crossfading. For more information on the granular synthesis process, check the *syncgrain* manual page.

## Syntax

```
asig syncloop kamp, kfreq, kpitch, kgrsize, kprate, klstart, \  
      klend, ifun1, ifun2, iolaps[,istart, iskip]
```

## Initialization

*ifun1* -- source signal function table. Deferred-allocation tables (see *GENOI*) are accepted, but the opcode expects a mono source.

*ifun2* -- grain envelope function table.

*iolaps* -- maximum number of overlaps,  $\max(kfreq) * \max(kgrsize)$ . Estimating a large value should not affect performance, but exceeding this value will probably have disastrous consequences.

*istart* -- starting point of synthesis in secs (defaults to 0).

*iskip* -- if 1, the opcode initialisation is skipped, for tied notes, performance continues from the position in the loop where the previous note stopped. The default, 0, does not skip initialisation

## Performance

*kamp* -- amplitude scaling

*kfreq* -- frequency of grain generation, or density, in grains/sec.

*kpitch* -- grain pitch scaling (1=normal pitch, < 1 lower, > 1 higher; negative, backwards)

*kgrsize* -- grain size in secs.

*kprate* -- readout pointer rate, in grains. The value of 1 will advance the reading pointer 1 grain ahead in the source table. Larger values will time-compress and smaller values will time-expand the source signal. Negative values will cause the pointer to run backwards and zero will freeze it.

*klstart* -- loop start in secs.

*klend* -- loop end in secs.

## Examples

Here is an example of the syncloop opcode. It uses the file *syncloop.csd* [examples/syncloop.csd].

### Example 965. Example of the syncloop opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o syncloop.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
Odbfs = 1
nchnls = 2

instr 1

iolaps = 2
igrsize = 0.01
ifreq = iolaps/igrsize
ips = 1/iolaps

istr = p4 /* timescale */
ipitch = 1 /* pitchscale */

asig syncloop 1, ifreq, ipitch, igrsize, ips*istr, .3, .75, 1, 2, iolaps
outs asig, asig

endin
</CsInstruments>
<CsScore>
f1 0 0 1 "beats.wav" 0 0 0
f2 0 8192 20 2 1

i1 0 6 .5
i1 7 6 .15
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
January 2005

New plugin in version 5

January 2005.

# syncphasor

syncphasor — Produces a normalized moving phase value with sync input and output.

## Description

Produces a moving phase value between zero and one and an extra impulse output ("sync out") whenever its phase value crosses or is reset to zero. The phase can be reset at any time by an impulse on the "sync in" parameter.

## Syntax

```
aphase, asyncout syncphasor xcps, asyncin, [, iphs]
```

## Initialization

*iphs* (optional) -- initial phase, expressed as a fraction of a cycle (0 to 1). A negative value will cause phase initialization to be skipped. The default value is zero.

## Performance

*aphase* -- the output phase value; always between 0 and 1.

*asyncout* -- the sync output has a value of 1.0 for one sample whenever the phase value crosses zero or whenever the sync input is non-zero. It is zero at all other times.

*asyncin* -- the sync input causes the phase to reset to zero whenever *asyncin* is non-zero.

*xcps* -- frequency of the phasor in cycles-per-second. If *xcps* is negative, the phase value will decrease from 1 to 0 instead of increasing.

An internal phase is successively accumulated in accordance with the *xcps* frequency to produce a moving phase value, normalized to lie in the range  $0 \leq \text{phs} < 1$ . When used as the index to a *table* unit, this phase (multiplied by the desired function table length) will cause it to behave like an oscillator.

The phase of *syncphasor* though can be synced to another phasor (or other signal) using the *asyncin* parameter. Any time that *asyncin* is a non-zero value, the value of *aphase* will be reset to zero. *syncphasor* also outputs its own "sync" signal that consists of a one-sample impulse whenever its phase crosses zero or is reset. This makes it easy to chain together multiple *syncphasor* opcodes to create an oscillator "hard sync" effect.

## Examples

Here is an example of the syncphasor opcode. It uses the file *syncphasor.csd* [examples/syncphasor.csd].

### Example 966. Example of the syncphasor opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
; Audio out  Audio in
```

```

-odac          -iadc    ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o abs.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

instr 1
; Use two syncphasors - one is the "master",
; the other the "slave"

; master's frequency determines pitch
imastercps =      cpspch(p4)
imaxamp     =      10000

; the slave's frequency affects the timbre
kslavecps   line   imastercps, p3, imastercps * 3

; the master "oscillator"
; the master has no sync input
anosync     init   0.0
am, async   syncphasor imastercps, anosync

; the slave "oscillator"
aout, as    syncphasor kslavecps, async

adeclick    linseg  0.0, 0.05, 1.0, p3 - 0.1, 1.0, 0.05, 0.0

; Output the slave's phase value which is a rising
; sawtooth wave. This produces aliasing, but hey, this
; this is just an example ;)

          out      aout * adeclick * imaxamp

endin

</CsInstruments>
<CsScore>

i1 0 1      7.00
i1 + 0.5    7.02
i1 + .      7.05
i1 + .      7.07
i1 + .      7.09
i1 + 2      7.06

e

</CsScore>
</CsoundSynthesizer>

```

Here is another example of the syncphasor opcode. It uses the file *syncphasor-CZresonance.csd* [examples/syncphasor-CZresonance.csd].

### Example 967. Another example of the syncphasor opcode.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out      Audio in      No messages
-odac          -iadc    -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o syncphasor-CZresonance.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>
; by Anthony Kozar. February 2008
; http://www.anthonykozar.net/

```

```

; Imitation of the Casio CZ-series synthesizer's "Resonance" waveforms
; using a synced phasor to read a sinusoid table. The jumps at the sync
; points are smoothed by multiplying with a windowing function controlled
; by the master phasor.

; Based on information from the Wikipedia article on phase distortion:
; http://en.wikipedia.org/wiki/Phase_distortion_synthesis

; Sawtooth Resonance waveform. Smoothing function is just the inverted
; master phasor.

; The Wikipedia article shows an inverted cosine as the stored waveform,
; which implies that it must be unipolar for the smoothing to work.
; I have substituted a sine wave in the first phrase to keep the output
; bipolar. The second phrase demonstrates the much "rezzier" sound of the
; bipolar cosine due to discontinuities.

instr 1
  ifreq      =      cpspch(p4)
  initReson  =      p5
  itable     =      p6
  imaxamp    =      10000
  anosync    init  0.0

  kslavecps  line    ifreq * initReson, p3, ifreq
  amaster, async syncphasor ifreq, anosync ; pair of phasors
  aslave, async2 syncphasor kslavecps, async ; slave synced to master
  aosc       tablei  aslave, itable, 1 ; use slave phasor to read a (co)sine table
  aout       =      aosc * (1.0 - amaster) ; inverted master smoothes jumps
  adeclick   linseg  0.0, 0.05, 1.0, p3 - 0.1, 1.0, 0.05, 0.0

          out      aout * adeclick * imaxamp
endin

; Triangle or Trapezoidal Resonance waveform. Uses a second table to change
; the shape of the smoothing function. (This is my best guess so far as to
; how these worked). The cosine table works fine with the triangular smoothing
; but we once again need to use a sine table with the trapezoidal smoothing.

; (It might be interesting to be able to vary the "width" of the trapezoid.
; This could be done with the pdhalf opcode).

instr 2
  ifreq      =      cpspch(p4)
  initReson  =      p5
  itable     =      p6
  ismoothtbl =      p7
  imaxamp    =      10000
  anosync    init  0.0

  kslavecps  line    ifreq * initReson, p3, ifreq
  amaster, async syncphasor ifreq, anosync ; pair of phasors
  aslave, async2 syncphasor kslavecps, async ; slave synced to master
  aosc       tablei  aslave, itable, 1 ; use slave phasor to read a (co)sine table
  asmooth    tablei  amaster, ismoothtbl, 1 ; use master phasor to read smoothing table
  aout       =      aosc * asmooth
  adeclick   linseg  0.0, 0.05, 1.0, p3 - 0.1, 1.0, 0.05, 0.0

  out      aout * adeclick * imaxamp
endin

</CsInstruments>
<CsScore>
f1 0 16385 10 1
f3 0 16385 9 1 1 270 ; inverted cosine
f5 0 4097 7 0.0 2048 1.0 2049 0.0 ; unipolar triangle

```

```
f6 0 4097 7 1.0 2048 1.0 2049 0.0 ; "trapezoid"

; Sawtooth resonance with a sine table
i1 0 1 7.00 5.0 1
i. + 0.5 7.02 4.0
i. + . 7.05 3.0
i. + . 7.07 2.0
i. + . 7.09 1.0
i. + 2 7.06 12.0
f0 6
s

; Sawtooth resonance with a cosine table
i1 0 1 7.00 5.0 3
i. + 0.5 7.02 4.0
i. + . 7.05 3.0
i. + . 7.07 2.0
i. + . 7.09 1.0
i. + 2 7.06 12.0
f0 6
s

; Triangle resonance with a cosine table
i2 0 1 7.00 5.0 3 5
i. + 0.5 7.02 4.0
i. + . 7.05 3.0
i. + . 7.07 2.0
i. + . 7.09 1.0
i. + 2 7.06 12.0
f0 6
s

; Trapezoidal resonance with a sine table
i2 0 1 7.00 5.0 1 6
i. + 0.5 7.02 4.0
i. + . 7.05 3.0
i. + . 7.07 2.0
i. + . 7.09 1.0
i. + 2 7.06 12.0

e

</CsScore>
</CsoundSynthesizer>
```

## See also

*phasor*.

And the *Table Access* opcodes like: *table*, *tablei*, *table3* and *tab*.

## Credits

Adapted from the *phasor* opcode by Anthony Kozar  
January 2008

New in Csound version 5.08

# system

`system` — Call an external program via the system call

## Description

`system` and `system_i` call any external command understood by the operating system, similarly to the C function `system()`. `system_i` runs at i-time only, while `system` runs both at initialization and performance time.

## Syntax

```
ires system_i itrig, Scmd, [inowait]
kres system ktrig, Scmd, [knowait]
```

## Initialization

*Scmd* -- command string

*itrig* -- if greater than zero the opcode executes the command; otherwise it is a null operation.

## Performance

*ktrig* -- if greater than zero and different from the value on the previous control cycle the opcode executes the command. Initially this previous value is taken as zero.

*inowait, knowait* -- if given a non zero the command is run in the background and the command does not wait for the result. (default = 0)

*ires, kres* -- the return code of the command in wait mode and if the command is run. In other cases returns zero.

More than one system command (a script) can be executed with a single `system` opcode by using double braces strings `{{ }}`.



### Note

This opcode is very system dependant, so should be used with extreme care (or not used) if platform neutrality is desired.

## Example

Here is an example of the `system_i` opcode. It uses the file `system.csd` [examples/system.csd].

### Example 968. Example of the system opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
```

```
; Select audio/midi flags here according to platform
; Audio out   Audio in
-odac          ; -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o system.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 1
; Waits for command to execute before continuing
ires system_i 1,{{      ps
    date
    cd ~/Desktop
    pwd
    ls -l
    whois csounds.com
}}
print ires
turnoff
endin

instr 2
; Runs command in a separate thread
ires system_i 1,{{      ps
    date
    cd ~/Desktop
    pwd
    ls -l
    whois csounds.com
}}, 1

print ires
turnoff
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for thirty seconds.
i 1 0 1
i 2 5 1
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitc  
2007

New in version 5.06

## tb

tb0, tb1, tb2, tb3, tb4, tb5, tb6, tb7, tb8, tb9, tb10, tb11, tb12, tb13, tb14, tb15, tb0\_init, tb1\_init, tb2\_init, tb3\_init, tb4\_init, tb5\_init, tb6\_init, tb7\_init, tb8\_init, tb9\_init, tb10\_init, tb11\_init, tb12\_init, tb13\_init, tb14\_init, tb15\_init — Table Read Access inside expressions.

## Description

Allow to read tables in function fashion, to be used inside expressions. At present time Csound only supports functions with a single input argument. However, to access table elements, user must provide two numbers, i.e. the number of table and the index of element. So, in order to allow to access a table element with a function, a previous preparation step should be done.

## Syntax

```
tb0_init ifn
tb1_init ifn
tb2_init ifn
tb3_init ifn
tb4_init ifn
tb5_init ifn
tb6_init ifn
tb7_init ifn
tb8_init ifn
tb9_init ifn
tb10_init ifn
tb11_init ifn
tb12_init ifn
tb13_init ifn
tb14_init ifn
tb15_init ifn

iout = tb0(iIndex)
kout = tb0(kIndex)

iout = tb1(iIndex)
kout = tb1(kIndex)

iout = tb2(iIndex)
kout = tb2(kIndex)

iout = tb3(iIndex)
kout = tb3(kIndex)

iout = tb4(iIndex)
```

```
kout = tb4(kIndex)
iout = tb5(iIndex)
kout = tb5(kIndex)
iout = tb6(iIndex)
kout = tb6(kIndex)
iout = tb7(iIndex)
kout = tb7(kIndex)
iout = tb8(iIndex)
kout = tb8(kIndex)
iout = tb9(iIndex)
kout = tb9(kIndex)
iout = tb10(iIndex)
kout = tb10(kIndex)
iout = tb11(iIndex)
kout = tb11(kIndex)
iout = tb12(iIndex)
kout = tb12(kIndex)
iout = tb13(iIndex)
kout = tb13(kIndex)
iout = tb14(iIndex)
kout = tb14(kIndex)
iout = tb15(iIndex)
kout = tb15(kIndex)
```

## Performance

There are 16 different opcodes whose name is associated with a number from 0 to 15. User can associate a specific table with each opcode (so the maximum number of tables that can be accessed in function fashion is 16). Prior to access a table, user must associate the table with one of the 16 opcodes by means of an opcode chosen among *tb0\_init*, ..., *tb15\_init*. For example,

```
tb0_init 1
```

associates table 1 with *tb0()* function, so that, each element of table 1 can be accessed (in function fashion) with:

```
kvar = tb0(k_some_index_of_table1) * k_some_other_var
```

```
ivar = tb0(i_some_index_of_table1) + i_some_other_var
```

etc...

By using these opcodes, user can drastically reduce the number of lines of an orchestra, improving its readability.

## Credits

Written by Gabriel Maldonado.

# tab

tab — Fast table opcodes.

## Description

Fast table opcodes. Faster than *table* and *tablew* because don't allow wrap-around and limit and don't check index validity. Have been implemented in order to provide fast access to arrays. Support non-power of two tables (can be generated by any GEN function by giving a negative length value).

## Syntax

```
ir tab_i indx, ifn[, ixmode]
kr tab kndx, ifn[, ixmode]
ar tab xndx, ifn[, ixmode]
tabw_i isig, indx, ifn [,ixmode]
tabw ksig, kndx, ifn [,ixmode]
tabw asig, andx, ifn [,ixmode]
```

## Initialization

*ifn* -- table number

*ixmode* -- defaults to zero. If zero *xndx* and *ixoff* ranges match the length of the table; if non zero *xndx* and *ixoff* have a 0 to 1 range.

*isig* -- input value to write.

*indx* -- table index

## Performance

*asig*, *ksig* -- input signal to write.

*andx*, *kndx* -- table index.

*tab* and *tabw* opcodes are similar to *table* and *tablew*, but are faster and support tables having non-power-of-two length.

Special care of index value must be taken into account. Index values out of the table allocated space will crash Csound.

## Examples

Here is an example of the tab opcode. It uses the file *tab.csd* [examples/tab.csd].

### Example 969. Example of the tab opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o tab.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gifn1 ftgen 1, 0, 0, 1, "flute.aiff", 0, 4, 0 ;deferred-size table

instr 1

atab init 0
isize tableng 1    ;length of table?
print isize
andx phasor 1 / (isize / sr)
asig tab andx, 1, 1    ;has a 0 to 1 range
outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 2.3
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado.

# tabifd

tabifd — Instantaneous Frequency Distribution, magnitude and phase analysis.

## Description

The tabifd opcode takes an input function table and performs an Instantaneous Frequency, magnitude and phase analysis, using the STFT and tabifd (Instantaneous Frequency Distribution), as described in Lazzarini et al, "Time-stretching using the Instantaneous Frequency Distribution and Partial Tracking", Proc.of ICMC05, Barcelona. It generates two PV streaming signals, one containing the amplitudes and frequencies (a similar output to pvsanal) and another containing amplitudes and unwrapped phases.

## Syntax

```
ffr,fpbs tabifd ktimpt, kamp, kpitch, ifftsize, ihopsize, iwintype,ifn
```

## Initialization

*ifftsize* -- FFT analysis size, must be power-of-two and integer multiple of the hopsize.

*ihopsize* -- hopsize in samples

*iwintype* -- window type (O: Hamming, 1: Hanning)

*ifn* -- source function table

## Performance

*ffr* -- output pv stream in AMP\_FREQ format

*fpbs* -- output pv stream in AMP\_PHASE format

*ktimpt* -- time point (in secs) to read from table (if less than 0 or bigger than table length, it will wraparound)

*kamp* -- amplitude scaling

*kpitch* -- pitch scaling (transposition)

## Examples

Here is an example of the tabifd opcode. It uses the file *tabifd.csd* [examples/tabifd.csd].

### Example 970. Example of the tabifd opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
```

```
; For Non-realtime ouput leave only the line below:
; -o tabifd.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
sr=44100
ksmps=1
nchnls=1
opcode TrackPlay, a, kkiiii
ktime,kthr,isiz,ihsiz,ifcos,ifn xin
idel = isiz-ihsiz*(isiz/(2*ihsiz)-1)
ffr,fphs tabifd ktime,10000,1, isiz, ihsiz, 1, ifn
ftrk partials ffr, fphs,kthr, 1, 1, 500
aout tradsyn ftrk, 2,1, 500, ifcos
xout aout
endop
instr 1
p3 = ftlen(2)/sr
ktime line 0,p3,p3
ares TrackPlay ktime, 0.003,2048,256,1,2
outs ares
endin

</CsInstruments>
<CsScore>
f1 0 16384 9 1 1 90
f2 0 0 1 "fox.wav" 0 0 1
i1 0 1
</CsScore>
</CsoundSynthesizer>
```

The example above shows the tabifd analysis feeding into partial tracking and cubic-phase additive resynthesis with pitch shifting.

## Credits

Author: Victor Lazzarini  
Aug 2015

New plugin in version 6

Aug 2015

# table

table — Accesses table values by direct indexing.

## Description

Accesses table values by direct indexing.

## Syntax

```
ares table andx, ifn [, ixmode] [, ixoff] [, iwrap]
ires table indx, ifn [, ixmode] [, ixoff] [, iwrap]
kres table kndx, ifn [, ixmode] [, ixoff] [, iwrap]
```

## Initialization

*ifn* -- function table number.

*ixmode* (optional) -- index data mode. The default value is 0.

- 0 = raw index
- 1 = normalized (0 to 1)

*ixoff* (optional) -- amount by which index is to be offset. For a table with origin at center, use `tablesize/2` (raw) or `.5` (normalized). The default value is 0.

*iwrap* (optional) -- wraparound index flag. The default value is 0.

- 0 = nowrap (index < 0 treated as index=0; index > tablesize sticks at index=size)
- 1 = wraparound.

## Performance

*table* invokes table lookup on behalf of init, control or audio indices. These indices can be raw entry numbers (0,1,2...size - 1) or scaled values (0 to 1-e). Indices are first modified by the offset value then checked for range before table lookup (see *iwrap*). If index is likely to be full scale, or if interpolation is being used, the table should have an extended guard point. *table* indexed by a periodic phasor ( see *phasor*) will simulate an oscillator.

## Examples

Here is an example of the table opcode. It uses the file *table.csd* [examples/table.csd].

### Example 971. Example of the table opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o table.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Vary our index linearly from 0 to 1.
kndx line 0, p3, 1

; Read Table #1 with our index.
ifn = 1
ixmode = 1
kfreq table kndx, ifn, ixmode

; Generate a sine waveform, use our table values
; to vary its frequency.
a1 oscil 20000, kfreq, 2
out a1
endin

</CsInstruments>
<CsScore>

; Table #1, a line from 200 to 2,000.
f 1 0 1025 -7 200 1024 2000
; Table #2, a sine wave.
f 2 0 16384 10 1

; Play Instrument #1 for 2 seconds.
i 1 0 2
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*tablei, table3, oscil1, oscilli, osciln*

## Credits

Example written by Kevin Conder.

## table3

`table3` — Accesses table values by direct indexing with cubic interpolation.

### Description

Accesses table values by direct indexing with cubic interpolation.

### Syntax

```
ares table3 andx, ifn [, ixmode] [, ixoff] [, iwrap]
ires table3 indx, ifn [, ixmode] [, ixoff] [, iwrap]
kres table3 kndx, ifn [, ixmode] [, ixoff] [, iwrap]
```

### Initialization

*ifn* -- function table number.

*ixmode* (optional) -- index data mode. The default value is 0.

- 0 = raw index
- 1 = normalized (0 to 1)

*ixoff* (optional) -- amount by which index is to be offset. For a table with origin at center, use `tablesize/2` (raw) or `.5` (normalized). The default value is 0.

*iwrap* (optional) -- wraparound index flag. The default value is 0.

- 0 = nowrap (index < 0 treated as index=0; index > tablesize sticks at index=size)
- 1 = wraparound.

### Performance

`table3` is identical to `tablei`, except that it uses cubic interpolation. (New in Csound version 3.50).

### See Also

`table`, `tablei`, `oscill`, `oscilli`, `osciln`

# tablecopy

tablecopy — Simple, fast table copy opcode.

## Description

Simple, fast table copy opcode.

## Syntax

```
tablecopy kdft, ksft
```

## Performance

*kdft* -- Destination function table.

*ksft* -- Number of source function table.

*tablecopy* -- Simple, fast table copy opcode. Takes the table length from the destination table, and reads from the start of the source table. For speed reasons, does not check the source length - just copies regardless - in “wrap” mode. This may read through the source table several times. A source table with length 1 will cause all values in the destination table to be written to its value.

*tablecopy* cannot read or write the guardpoint. To read it use *table*, with *ndx* = the table length. Likewise use *table* write to write it.

To write the guardpoint to the value in location 0, use *tablegpw*.

This is primarily to change function tables quickly in a real-time situation.

## Examples

Here is an example of the tablecopy opcode. It uses the file *tablecopy.csd* [examples/tablecopy.csd].

### Example 972. Example of the tablecopy opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o tablecopy.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
;after an example from Jonathan Murphy
```

```

gilen = 7
gilist ftgen 1, 0, gilen + 1, -2, 2, 3, 4, 5, 6, 7 ;only 6 elements, so 1 is doubled
gitmp ftgen 2, 0, gilen + 1, -2, 0 ;empty table
gkmax init gilen

seed 0 ;each time different

instr 1

ktrig metro 3 ;trigger values
krnd random 0, gkmax

if (ktrig == 1) then
  kval table krnd, gilist
  tablew 0, krnd, gilist
  kread = 0
  kwrite = 0
start:
  knew table kread, gilist
if (knew != 0) then
  tablew knew, kwrite, gitmp
  kwrite = kwrite + 1
endif
  kread = kread + 1
if (kread <= gilen) kgoto start
  tablecopy gilist, gitmp ;fill with zeroes
  gkmax = gkmax - 1
endif

printk2 kval

if (gkmax < 0) then
  event "i", 2, 0, 1/kr ;when ready, then stop
endif

asig vco2 .5, 40*kval ;sound generation
outs asig, asig

endin

instr 2

exitnow

endin
</CsInstruments>
<CsScore>
i1 0 5
e
</CsScore>
</CsSoundSynthesizer>

```

Its output should include a lines like these:

```

i1 5.00000
i1 3.00000
i1 2.00000
i1 4.00000
i1 7.00000
i1 6.00000
i1 7.00000

```

## See Also

*tablegpw, tablemix, tablecopy, tableigpw, tableimix*

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.47

# tablefilter

tablefilter — Filters a source table and writes result into a destination table.

## Description

This opcode can be used in order to filter values from function tables following certain algorithms. The filtered output is written into a destination table and the number of elements that have passed the filter is returned.

## Syntax

```
knumpassed tablefilter kouttable, kintatable, kmode, kparam
```

## Performance

*knumpassed* -- the number of elements that have passed the filter.

*kouttable* -- the number of the table containing the values that have passed.

*kintatable* -- the number of the table used as filter input.

*kmode* -- mode of the filter:

- 1 -- tests the weight of the denominators of the fractions in the source table. Letting pass only values from the source that are less heavy than the weight of the threshold.
- 2 -- tests the weight of the denominators of the fractions in the source table. Letting pass only values from the source that are heavier than or equal to the weight of the threshold.

*kparam* -- integer threshold parameter for the filter. It means that denominators whose weights are heavier than the weight of this threshold are not passed through the filter. The weight of an integer is calculated using Clarence Barlow's function of indigestibility of a number. According to this function, higher prime numbers contribute to an increased weight of any natural integer they divide. The order of the first 16 integers according to their indigestibility is: 1, 2, 4, 3, 8, 6, 16, 12, 9, 5, 10, 15, 7, 14.

## Examples

Here is an example of the tablefilter opcode. It uses the file *tablefilter.csd* [examples/tablefilter.csd].

### Example 973. Example of the tablefilter opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
  
</CsOptions>  
<CsInstruments>  
  
sr=44100  
ksmps=10  
nchnls=1
```

```

gifarn init 8 ; initialise integer for Farey Sequence F_8
gires fareyleni gifarn ; calculate length of F_8, returns 23
; the table length won't be a power of 2
; (The length of a Farey Sequence with n > 1 is always odd)
gilen init gires * -1

gifarey ftgen 200, 0, gilen, "farey", gifarn, 0

; initialize destination table with 0s
gifiltered ftgen 0, 0, gilen, 21, 1, 0

; initialize second destination table with 0s
gifiltered2 ftgen 0, 0, gilen, 21, 1, 0

; table filtering opcode: dest. source, mode, threshold
ginumpassed tablefilteri gifiltered, gifarey, 1, 6
; the threshold parameter indicates that denominators whose weights are heavier
; than 6 are not passing through the filter. The weight is calculated using
; Clarence Barlow's function of indigestibility of a number. According to this function,
; higher prime numbers contribute to an increased weight of any natural integer they divide.
; ginumpassed is the number of elements from the source table 'gifarey'
; that have passed the test and which have been copied to the destination table 'gifiltered'

; apply a different filter:
ginumpassed2 tablefilteri gifiltered2, gifarey, 2, 5
; In mode=2 we again test the digestibility of the denominators of the
; fractions in the source table.
; The difference to mode=1 is that we now let pass only vaules from the
; source that are as heavy as the threshold or greater.

instr 4
kndx init 0 ; read out elements of now filtered F_8 sequentially and print to file
if (kndx < ginumpassed) then
kelem tab kndx, gifiltered
fprintks "fareyfilter_lp.txt", "%2.6f\\n", kelem
kndx = kndx+1
endif
endin

instr 5
kndx init 0 ; read out elements and print to file
if (kndx < ginumpassed2) then
kelem tab kndx, gifiltered2
fprintks "fareyfilter_hp.txt", "%2.6f\\n", kelem
kndx = kndx+1
endif
endin

</CsInstruments>
<CsScore>

i4 0 1
i5 0 1
e
</CsScore>
</CsoundSynthesizer>

```

## Credits

Author: Georg Boenn  
University of Glamorgan, UK

New in Csound version 5.13

# tablefilteri

tablefilteri — Filters a source table and writes result into a destination table.

## Description

This opcode can be used in order to filter values from function tables following certain algorithms. The filtered output is written into a destination table and the number of elements that have passed the filter is returned.

## Syntax

```
inumpassed tablefilteri iouttable, iintatable, imode, iparam
```

## Initialization

*inumpassed* -- the number of elements that have passed the filter.

*iouttable* -- the number of the table containing the values that have passed.

*iintatable* -- the number of the table used as filter input.

*imode* -- mode of the filter:

- 1 -- tests the weight of the denominators of the fractions in the source table. Letting pass only values from the source that are less heavy than the weight of the threshold.
- 2 -- tests the weight of the denominators of the fractions in the source table. Letting pass only values from the source that are heavier than or equal to the weight of the threshold.

*iparam* -- integer threshold parameter for the filter. It means that denominators whose weights are heavier than the weight of this threshold are not passed through the filter. The weight of an integer is calculated using Clarence Barlow's function of indigestibility of a number. According to this function, higher prime numbers contribute to an increased weight of any natural integer they divide. The order of the first 16 integers according to their indigestibility is: 1, 2, 4, 3, 8, 6, 16, 12, 9, 5, 10, 15, 7, 14.

## Examples

Here is an example of the tablefilteri opcode. It uses the file *tablefilter.csd* [examples/tablefilter.csd].

### Example 974. Example of the tablefilteri opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>

</CsOptions>
<CsInstruments>

sr=44100
ksmps=10
nchnls=1
```

```

gifarn init 8 ; initialise integer for Farey Sequence F_8
gires fareyleni gifarn ; calculate length of F_8, returns 23
; the table length won't be a power of 2
; (The length of a Farey Sequence with n > 1 is always odd)
gilen init gires * -1

gifarey ftgen 200, 0, gilen, "farey", gifarn, 0

; initialize destination table with 0s
gifiltered ftgen 0, 0, gilen, 21, 1, 0

; initialize second destination table with 0s
gifiltered2 ftgen 0, 0, gilen, 21, 1, 0

; table filtering opcode: dest. source, mode, threshold
ginumpassed tablefilteri gifiltered, gifarey, 1, 6
; the threshold parameter indicates that denominators whose weights are heavier
; than 6 are not passing through the filter. The weight is calculated using
; Clarence Barlow's function of indigestibility of a number. According to this function,
; higher prime numbers contribute to an increased weight of any natural integer they divide.
; ginumpassed is the number of elements from the source table 'gifarey'
; that have passed the test and which have been copied to the destination table 'gifiltered'

; apply a different filter:
ginumpassed2 tablefilteri gifiltered2, gifarey, 2, 5
; In mode=2 we again test the digestibility of the denominators of the
; fractions in the source table.
; The difference to mode=1 is that we now let pass only vaules from the
; source that are as heavy as the threshold or greater.

instr 4
kndx init 0 ; read out elements of now filtered F_8 sequentially and print to file
if (kndx < ginumpassed) then
kelem tab kndx, gifiltered
fprintks "fareyfilter_lp.txt", "%2.6f\\n", kelem
kndx = kndx+1
endif
endin

instr 5
kndx init 0 ; read out elements and print to file
if (kndx < ginumpassed2) then
kelem tab kndx, gifiltered2
fprintks "fareyfilter_hp.txt", "%2.6f\\n", kelem
kndx = kndx+1
endif
endin

</CsInstruments>
<CsScore>

i4 0 1
i5 0 1
e
</CsScore>
</CsoundSynthesizer>

```

## Credits

Author: Georg Boenn  
University of Glamorgan, UK

New in Csound version 5.13

# tablegpw

tablegpw — Writes a table's guard point.

## Description

Writes a table's guard point.

## Syntax

```
tablegpw kfn
```

## Performance

*kfn* -- Table number to be interrogated

*tablegpw* -- For writing the table's guard point, with the value which is in location 0. Does nothing if table does not exist.

Likely to be useful after manipulating a table with *tablemix* or *tablecopy*.

## See Also

*tablecopy*, *tablemix*, *tablecopy*, *tableigpw*, *tableimix*

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.47

# tablei

tablei — Accesses table values by direct indexing with linear interpolation.

## Description

Accesses table values by direct indexing with linear interpolation.

## Syntax

```
ares tablei andx, ifn [, ixmode] [, ixoff] [, iwrap]
ires tablei indx, ifn [, ixmode] [, ixoff] [, iwrap]
kres tablei kndx, ifn [, ixmode] [, ixoff] [, iwrap]
```

## Initialization

*ifn* -- function table number. *tablei* requires the extended guard point.

*ixmode* (optional) -- index data mode. The default value is 0.

- 0 = raw index
- 1 = normalized (0 to 1)

*ixoff* (optional) -- amount by which index is to be offset. For a table with origin at center, use *tablesize/2* (raw) or *.5* (normalized). The default value is 0.

*iwrap* (optional) -- wraparound index flag. The default value is 0.

- 0 = nowrap (index < 0 treated as index=0; index > *tablesize* sticks at index=size)
- 1 = wraparound.

## Performance

*tablei* is a interpolating unit in which the fractional part of index is used to interpolate between adjacent table entries. The smoothness gained by interpolation is at some small cost in execution time (see also *oscili*, etc.), but the interpolating and non-interpolating units are otherwise interchangeable. Note that when *tablei* uses a periodic index whose modulo *n* is less than the power of 2 table length, the interpolation process requires that there be an (*n* + 1)th table value that is a repeat of the 1st (see *f Statement* in score).

## Examples

Here is an example of the *tablei* opcode. It uses the file *tablei.csd* [examples/tablei.csd].

### Example 975. Example of the *tablei* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;;realtime audio out
```

```

;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o tablei.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

seed 0 ;generate new values every time the instr is played

instr 1

ifn = p4
isize = p5
ithresh = 0.5

itemp ftgen ifn, 0, isize, 21, 2

iwrite_value = 0
i_index = 0

loop_start:
  iread_value tablei i_index, ifn

  if iread_value > ithresh then
    iwrite_value = 1
  else
    iwrite_value = -1
  endif
tableiw iwrite_value, i_index, ifn
loop_lt i_index, 1, isize, loop_start
turnoff

endin

instr 2

ifn = p4
isize = ftlen(ifn)
prints "Index\tValue\n"

i_index = 0
loop_start:
  ivalue tablei i_index, ifn
  prints "%d:\t%f\n", i_index, ivalue

  loop_lt i_index, 1, isize, loop_start ;read table 1 with our index

aout oscili .5, 100, ifn ;use table to play the polypulse
outs aout, aout

endin
</CsInstruments>
<CsScore>
i 1 0 1 100 16
i 2 0 2 100
e
</CsScore>
</CsSoundSynthesizer>

```

## See Also

*table, table3, oscil, oscilli, osciln*

More information on this opcode: <http://www.csoundjournal.com/issue12/genInstruments.html> , written by Jacob Joaquin

# tablecopy

`tablecopy` — Simple, fast table copy opcode.

## Description

Simple, fast table copy opcode.

## Syntax

```
tablecopy idft, isft
```

## Initialization

*idft* -- Destination function table.

*isft* -- Number of source function table.

## Performance

*tablecopy* -- Simple, fast table copy opcodes. Takes the table length from the destination table, and reads from the start of the source table. For speed reasons, does not check the source length - just copies regardless - in "wrap" mode. This may read through the source table several times. A source table with length 1 will cause all values in the destination table to be written to its value.

*tablecopy* cannot read or write the guardpoint. To read it use *table*, with *ndx* = the table length. Likewise use *tablewrite* to write it.

To write the guardpoint to the value in location 0, use *tablegpw*.

This is primarily to change function tables quickly in a real-time situation.

## See Also

*tablecopy*, *tablegpw*, *tablemix*, *tableigpw*, *tableimix*

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.47

# tableigpw

`tableigpw` — Writes a table's guard point.

## Description

Writes a table's guard point.

## Syntax

```
tableigpw ifn
```

## Initialization

*ifn* -- Table number to be interrogated

## Performance

*tableigpw* -- For writing the table's guard point, with the value which is in location 0. Does nothing if table does not exist.

Likely to be useful after manipulating a table with *tablemix* or *tablecopy*.

## See Also

*tablecopy*, *tablegpw*, *tablemix*, *tableicopy*, *tableimix*

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.47

# tableikt

tableikt — Provides k-rate control over table numbers.

## Description

k-rate control over table numbers. Function tables are read with linear interpolation.

The standard Csound opcode *tablei*, when producing a k- or a-rate result, can only use an init-time variable to select the table number. *tableikt* accepts k-rate control as well as i-time. In all other respects they are similar to the original opcodes.

## Syntax

```
ares tableikt xndx, kfn [, ixmode] [, ixoff] [, iwrap]
```

```
kres tableikt kndx, kfn [, ixmode] [, ixoff] [, iwrap]
```

## Initialization

*ixmode* -- if 0, *xndx* and *ixoff* ranges match the length of the table. if non-zero *xndx* and *ixoff* have a 0 to 1 range. Default is 0

*ixoff* -- if 0, total index is controlled directly by *xndx*, ie. the indexing starts from the start of the table. If non-zero, start indexing from somewhere else in the table. Value must be positive and less than the table length (*ixmode* = 0) or less than 1 (*ixmode* not equal to 0). Default is 0.

*iwrap* -- if *iwrap* = 0, *Limit mode*: when total index is below 0, then final index is 0. Total index above table length results in a final index of the table length - high out of range total indexes stick at the upper limit of the table. If *iwrap* not equal to 0, *Wrap mode*: total index is wrapped modulo the table length so that all total indexes map into the table. For instance, in a table of length 8, *xndx* = 5 and *ixoff* = 6 gives a total index of 11, which wraps to a final index of 3. Default is 0.

## Performance

*kndx* -- Index into table, either a positive number range matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* not equal to 0).

*xndx* -- matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* not equal to 0)

*kfn* -- Table number. Must be  $\geq 1$ . Floats are rounded down to an integer. If a table number does not point to a valid table, or the table has not yet been loaded (*GENOI*) then an error will result and the instrument will be de-activated.



### Caution with k-rate table numbers

At k-rate, if a table number of  $< 1$  is given, or the table number points to a non-existent table, or to one which has a length of 0 (it is to be loaded from a file later) then an error will result and the instrument will be deactivated. *kfn* must be initialized at the appropriate rate using *init*. Attempting to load an i-rate value into *kfn* will result in an error.

## Examples

Here is an example of the `tableikt` opcode. It uses the file `tableikt.csd` [examples/tableikt.csd].

### Example 976. Example of the `tableikt` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o tableikt.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ilowfn = p4      ;lowest ftable wave
ihighfn = p5     ;highest ftable wave

kswpenv line 1, p3, 0      ;sweep envelope, calculate current table pair and interpolation amount
inumtables = ihighfn - ilowfn ;1 less than number of tables
kfn1 = int(kswpenv*inumtables) + ilowfn
      printks "play table no: %d\n", 1, kfn1
kfn2 = kfn1 + 1
kinterp = frac(kswpenv*inumtables)
ixmode = 1      ;read tables with phasor
aphase phasor 40
asig tableikt aphase, kfn1, ixmode ;normalized index
      if kswpenv == 1.0 kgoto skipfn2 ;if kfn1 is last table, there is no kfn2
asig2 tableikt aphase, kfn2, ixmode
      skipfn2:
      amix ntrpol asig, asig2, kinterp ;interpolate between tables and output
outs amix*.5, amix*.5

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1
f 2 0 16384 10 1 .5
f 3 0 16384 1 "fox.wav" 0 0 0 ;a sample
f 4 0 16384 10 1 .5 .3 .25 .2 .16 .14 .125 .111 ;sawtooth
f 5 0 16384 10 1 .4 .3 .25 .2
f 6 0 16384 10 1 .3 .3 .25 .2 .16
f 7 0 16384 10 1 1 1 1 .7 .5 .3 .1 ;pulse
f 8 0 16384 1 "beats.wav" 0 0 0 ;a sample

i 1 0 10 1 8
e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like these:

```
play table no: 8  
play table no: 7  
play table no: 6  
.....  
play table no: 2  
play table no: 1
```

## See Also

*tablekt*

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.47

# tablemix

tablemix — Mixes two tables.

## Description

Mixes two tables.

## Syntax

```
tablemix idft, idoff, ilen, is1ft, is1off, is1g, is2ft, is2off, is2g
```

## Initialization

*idft* -- Destination function table.

*idoff* -- Offset to start writing from. Can be negative.

*ilen* -- Number of write operations to perform. Negative means work backwards.

*is1ft, is2ft* -- Source function tables. These can be the same as the destination table, if care is exercised about direction of copying data.

*is1off, is2off* -- Offsets to start reading from in source tables.

*is1g, is2g* -- Gains to apply when reading from the source tables. The results are added and the sum is written to the destination table.

## Performance

*tablemix* -- This opcode mixes from two tables, with separate gains into the destination table. Writing is done for *ilen* locations, usually stepping forward through the table - if *ilen* is positive. If it is negative, then the writing and reading order is backwards - towards lower indexes in the tables. This bi-directional option makes it easy to shift the contents of a table sideways by reading from it and writing back to it with a different offset.

If *ilen* is 0, no writing occurs. Note that the internal integer value of *ilen* is derived from the ANSI C floor() function - which returns the next most negative integer. Hence a fractional negative *ilen* value of -2.3 would create an internal length of 3, and cause the copying to start from the offset locations and proceed for two locations to the left.

The total index for table reading and writing is calculated from the starting offset for each table, plus the index value, which starts at 0 and then increments (or decrements) by 1 as mixing proceeds.

These total indexes can potentially be very large, since there is no restriction on the offset or the *ilen*. However each total index for each table is ANDed with a length mask (such as 0000 0111 for a table of length 8) to form a final index which is actually used for reading or writing. So no reading or writing can occur outside the tables. This is the same as “wrap” mode in table read and write. These opcodes do not read or write the guardpoint. If a table has been rewritten with one of these, then if it has a guardpoint which is supposed to contain the same value as the location 0, then call *tableigpw* afterwards.

The indexes and offsets are all in table steps - they are not normalized to 0 - 1. So for a table of length 256, *ilen* should be set to 256 if all the table was to be read or written.

The tables do not need to be the same length - wrapping occurs individually for each table.

## Examples

Here is an example of the `tablemix` opcode. It uses the file `tablemix.csd` [examples/tablemix.csd].

### Example 977. Example of the `tablemix` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o tablemix.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gisine ftgen 1, 0, 256, 10, 1, 0, 0, .4 ;sinoid
gisaw  ftgen 2, 0, 1024, 7, 0, 256, 1 ;saw
gimix  ftgen 100, 0, 256, 7, 0, 256, 1 ;used to mix

instr 1

    tablemix 100, 0, 256, 1, 0, 1, 2, 0, .5
    asig  poscil .5, 110, gimix ;mix table 1 & 2
    outs  asig, asig

endin
</CsInstruments>
<CsScore>

i1 0 5
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*tablecopy*, *tablegpw*, *tablemix*, *tableicopy*, *tableigpw*

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.47

# tableiw

tableiw — Change the contents of existing function tables.

## Description

This opcode operates on existing function tables, changing their contents. *tableiw* is used when all inputs are init time variables or constants and you only want to run it at the initialization of the instrument. The valid combinations of variable types are shown by the first letter of the variable names.

## Syntax

```
tableiw isig, indx, ifn [, ixmode] [, ixoff] [, iwgmode]
```

## Initialization

*isig* -- Input value to write to the table.

*indx* -- Index into table, either a positive number range matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* not equal to 0)

*ifn* -- Table number. Must be  $\geq 1$ . Floats are rounded down to an integer. If a table number does not point to a valid table, or the table has not yet been loaded (*GENOI*) then an error will result and the instrument will be de-activated.

*ixmode* (optional, default=0) -- index mode.

- 0 = *indx* and *ixoff* ranges match the length of the table.
- not equal to 0 = *indx* and *ixoff* have a 0 to 1 range.

*ixoff* (optional, default=0) -- index offset.

- 0 = Total index is controlled directly by *indx*, i.e. the indexing starts from the start of the table.
- Not equal to 0 = Start indexing from somewhere else in the table. Value must be positive and less than the table length (*ixmode* = 0) or less than 1 (*ixmode* not equal to 0).

*iwgmode* (optional, default=0) -- Wrap and guard point mode.

- 0 = Limit mode.
- 1 = Wrap mode.
- 2 = Guardpoint mode.

## Performance

### Limit mode (0)

Limit the total index (*indx* + *ixoff*) to between 0 and the guard point. For a table of length 5, this means that locations 0 to 3 and location 4 (the guard point) can be written. A negative total index writes to location 0.

## Wrap mode (1)

Wrap total index value into locations 0 to E, where E is either one less than the table length or the factor of 2 number which is one less than the table length. For example, wrap into a 0 to 3 range - so that total index 6 writes to location 2.

## Guardpoint mode (2)

The guardpoint is written at the same time as location 0 is written - with the same value.

This facilitates writing to tables which are intended to be read with interpolation for producing smooth cyclic waveforms. In addition, before it is used, the total index is incremented by half the range between one location and the next, before being rounded down to the integer address of a table location.

Normally (*igwmode* = 0 or 1) for a table of length 5 - which has locations 0 to 3 as the main table and location 4 as the guard point, a total index in the range of 0 to 0.999 will write to location 0. ("0.999" means just less than 1.0.) 1.0 to 1.999 will write to location 1 etc. A similar pattern holds for all total indexes 0 to 4.999 (*igwmode* = 0) or to 3.999 (*igwmode* = 1). *igwmode* = 0 enables locations 0 to 4 to be written - with the guardpoint (4) being written with a potentially different value from location 0.

With a table of length 5 and the *igwmode* = 2, then when the total index is in the range 0 to 0.499, it will write to locations 0 and 4. Range 0.5 to 1.499 will write to location 1 etc. 3.5 to 4.0 will also write to locations 0 and 4.

This way, the writing operation most closely approximates the results of interpolated reading. Guard point mode should only be used with tables that have a guardpoint.

Guardpoint mode is accomplished by adding 0.5 to the total index, rounding to the next lowest integer, wrapping it modulo the factor of two which is one less than the table length, writing the table (locations 0 to 3 in our example) and then writing to the guard point if index = 0.

## Examples

Here is an example of the `tableiw` opcode. It uses the file `tableiw.csd` [examples/tableiw.csd].

### Example 978. Example of the `tableiw` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o tableiw.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

seed 0 ;generate new values every time the instr is played

instr 1
```

```
ifn = p4
isize = p5
ithresh = 0.5

itemp ftgen ifn, 0, isize, 21, 2

iwrite_value = 0
i_index = 0

loop_start:
  iread_value tablei i_index, ifn

  if iread_value > ithresh then
    iwrite_value = 1
  else
    iwrite_value = -1
  endif
tableiw iwrite_value, i_index, ifn
loop_lt i_index, 1, isize, loop_start
turnoff

endin

instr 2

ifn = p4
isize = ftlen(ifn)
prints "Index\tValue\n"

i_index = 0
loop_start:
  ivalue tablei i_index, ifn
  prints "%d:\t%f\n", i_index, ivalue

  loop_lt i_index, 1, isize, loop_start ;read table 1 with our index

aout oscili .5, 100, ifn ;use table to play the polypulse
outs aout, aout

endin
</CsInstruments>
<CsScore>
i 1 0 1 100 16
i 2 0 2 100
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*tablew, tablewkt*

More information on this opcode: <http://www.csoundjournal.com/issue12/genInstruments.html> , written by Jacob Joaquin

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.47

Updated August 2002, thanks go to Abram Hindle for pointing out the correct syntax.

# tablekt

tablekt — Provides k-rate control over table numbers.

## Description

k-rate control over table numbers.

The standard Csound opcode *table* when producing a k- or a-rate result, can only use an init-time variable to select the table number. *tablekt* accepts k-rate control as well as i-time. In all other respects they are similar to the original opcodes.

## Syntax

```
ares tablekt xndx, kfn [, ixmode] [, ixoff] [, iwrap]
```

```
kres tablekt kndx, kfn [, ixmode] [, ixoff] [, iwrap]
```

## Initialization

*ixmode* -- if 0, *xndx* and *ixoff* ranges match the length of the table. if non-zero *xndx* and *ixoff* have a 0 to 1 range. Default is 0

*ixoff* -- if 0, total index is controlled directly by *xndx*, ie. the indexing starts from the start of the table. If non-zero, start indexing from somewhere else in the table. Value must be positive and less than the table length (*ixmode* = 0) or less than 1 (*ixmode* not equal to 0). Default is 0.

*iwrap* -- if *iwrap* = 0, *Limit mode*: when total index is below 0, then final index is 0. Total index above table length results in a final index of the table length - high out of range total indexes stick at the upper limit of the table. If *iwrap* not equal to 0, *Wrap mode*: total index is wrapped modulo the table length so that all total indexes map into the table. For instance, in a table of length 8, *xndx* = 5 and *ixoff* = 6 gives a total index of 11, which wraps to a final index of 3. Default is 0.

## Performance

*kndx* -- Index into table, either a positive number range matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* not equal to 0).

*xndx* -- matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* not equal to 0)

*kfn* -- Table number. Must be  $\geq 1$ . Floats are rounded down to an integer. If a table number does not point to a valid table, or the table has not yet been loaded (*GENOI*) then an error will result and the instrument will be de-activated.



### Caution with k-rate table numbers

At k-rate, if a table number of  $< 1$  is given, or the table number points to a non-existent table, or to one which has a length of 0 (it is to be loaded from a file later) then an error will result and the instrument will be deactivated. *kfn* must be initialized at the appropriate rate using *init*. Attempting to load an i-rate value into *kfn* will result in an error.

## Examples

Here is an example of the `tablekt` opcode. It uses the file `tablekt.csd` [examples/tablekt.csd].

### Example 979. Example of the `tablekt` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o tablekt.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
Odbfs = 1

seed 0

gift1 ftgen 1, 0, 1024, 10, 1 ;sine wave
gift2 ftgen 2, 0, 1024, 10, 1, 1, 1, 1, 1, 1, 1, 1, 1 ;pulse

instr 1
andx phasor 400 ;phasor for reading the index
kfn init 1 ;initialize the choice of the function table
kmetro init 1 ;initialize the frequency of the metro
knewft metro kmetro ;make a new choice for selecting the function table once a second

if knewft == 1 then
  kfn = (kfn == 1 ? 2 : 1) ;switch between 1 and 2
  kmetro random .5, 2 ;create new metro frequency
  printk2 kfn
endif

ares tablekt andx, kfn, 1
outs ares, ares
endin

</CsInstruments>
<CsScore>
i 1 0 10
e
</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like these:

```

i1      2.00000
i1      1.00000
i1      2.00000
i1      1.00000
....

```

## See Also

*tableikt*

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.47

# tablemix

tablemix — Mixes two tables.

## Description

Mixes two tables.

## Syntax

```
tablemix kdft, kdoff, klen, ks1ft, ks1off, ks1g, ks2ft, ks2off, ks2g
```

## Performance

*kdft* -- Destination function table.

*kdoff* -- Offset to start writing from. Can be negative.

*klen* -- Number of write operations to perform. Negative means work backwards.

*ks1ft, ks2ft* -- Source function tables. These can be the same as the destination table, if care is exercised about direction of copying data.

*ks1off, ks2off* -- Offsets to start reading from in source tables.

*ks1g, ks2g* -- Gains to apply when reading from the source tables. The results are added and the sum is written to the destination table.

*tablemix* -- This opcode mixes from two tables, with separate gains into the destination table. Writing is done for *klen* locations, usually stepping forward through the table - if *klen* is positive. If it is negative, then the writing and reading order is backwards - towards lower indexes in the tables. This bi-directional option makes it easy to shift the contents of a table sideways by reading from it and writing back to it with a different offset.

If *klen* is 0, no writing occurs. Note that the internal integer value of *klen* is derived from the ANSI C floor() function - which returns the next most negative integer. Hence a fractional negative *klen* value of -2.3 would create an internal length of 3, and cause the copying to start from the offset locations and proceed for two locations to the left.

The total index for table reading and writing is calculated from the starting offset for each table, plus the index value, which starts at 0 and then increments (or decrements) by 1 as mixing proceeds.

These total indexes can potentially be very large, since there is no restriction on the offset or the *klen*. However each total index for each table is ANDed with a length mask (such as 0000 0111 for a table of length 8) to form a final index which is actually used for reading or writing. So no reading or writing can occur outside the tables. This is the same as “wrap” mode in table read and write. These opcodes do not read or write the guardpoint. If a table has been rewritten with one of these, then if it has a guardpoint which is supposed to contain the same value as the location 0, then call *tablegpw* afterwards.

The indexes and offsets are all in table steps - they are not normalized to 0 - 1. So for a table of length 256, *klen* should be set to 256 if all the table was to be read or written.

The tables do not need to be the same length - wrapping occurs individually for each table.

## Examples

Here is an example of the `tablemix` opcode. It uses the file `tablemix.csd` [examples/tablemix.csd].

### Example 980. Example of the `tablemix` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o tablemix.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gisinoid ftgen 1, 0, 256, 10, 1, 0, 0, .4 ;sinoid
gisaw    ftgen 2, 0, 1024, 7, 0, 256, 1 ;saw
gimix    ftgen 100, 0, 256, 7, 0, 256, 1 ;destination table

instr 1

kgain linseg 0, p3*.5, .5, p3*.5, 0
tablemix 100, 0, 256, 1, 0, 1, 2, 0, kgain
asig    poscil .5, 110, gimix ;mix table 1 & 2
outs    asig, asig

endin
</CsInstruments>
<CsScore>

i1 0 10

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*tablecopy*, *tablegpw*, *tableicopy*, *tableigpw*, *tableimix*

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.47

# tableng

tableng — Interrogates a function table for length.

## Description

Interrogates a function table for length.

## Syntax

```
ires tableng ifn
kres tableng kfn
```

## Initialization

*ifn* -- Table number to be interrogated

## Performance

*kfn* -- Table number to be interrogated

*tableng* returns the length of the specified table. This will be a power of two number in most circumstances. It will not show whether a table has a guardpoint or not. It seems this information is not available in the table's data structure. If the specified table is not found, then 0 will be returned.

Likely to be useful for setting up code for table manipulation operations, such as *tablemix* and *tablecopy*.

## Examples

Here is an example of the tableng opcode. It uses the file *tableng.csd* [examples/tableng.csd].

### Example 981. Example of the tableng opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o tableng.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gifn1 ftgen 1, 0, 0, 1, "flute.aiff", 0, 4, 0 ;deferred-size table
```

```
instr 1

isize tableng 1
print isize
andx phasor 1 / (isize / sr) ;play at correct pitch
asig tab andx, 1, 1
      outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 2.3
e
</CsScore>
</CsoundSynthesizer>
```

Its output should include a line like this:

```
instr 1:  isize = 115506.000
```

## Credits

Author: Robin Whittle  
Australia  
May 1997

# tablera

tablera — Reads tables in sequential locations.

## Description

These opcode reads tables in sequential locations to an a-rate variable. Some thought is required before using it. It has at least two major, and quite different, applications which are discussed below.

## Syntax

```
ares tablera kfn, kstart, koff
```

## Performance

*ares* -- a-rate destination for reading *ksmps* values from a table.

*kfn* -- i- or k-rate number of the table to read or write.

*kstart* -- Where in table to read or write.

*koff* -- i- or k-rate offset into table. Range unlimited - see explanation at end of this section.

In one application, *tablera* is intended to be used in pair with *tablewa*, or with several *tablera* opcodes before a *tablewa* -- all sharing the same *kstart* variable.

These read from and write to sequential locations in a table at audio rates, with *ksmps* floats being written and read each cycle.

*tablera* starts reading from location *kstart*. *tablewa* starts writing to location *kstart*, and then writes to *kstart* with the number of the location one more than the one it last wrote. (Note that for *tablewa*, *kstart* is both an input and output variable.) If the writing index reaches the end of the table, then no further writing occurs and zero is written to *kstart*.

For instance, if the table's length was 16 (locations 0 to 15), and *ksmps* was 5. Then the following steps would occur with repetitive runs of the *tablewa* opcode, assuming that *kstart* started at 0.

Run Number	Initial kstart	Final kstart	Locations Written
1	0	5	0 1 2 3 4
2	5	10	5 6 7 8 9
3	10	15	10 11 12 13 14
4	15	0	15

This is to facilitate processing table data using standard a-rate orchestra code between the *tablera* and *tablewa* opcodes. They allow all Csound k-rate operators to be used (with caution) on a-rate variables - something that would only be possible otherwise by *ksmps* = 1, *downsamp* and *upsamp*.



### Several cautions

- The k-rate code in the processing loop is really running at a-rate, so time dependent functions like *port* and *oscil* work faster than normal - their code is expecting to be running at k-rate.

- This system will produce undesirable results unless the *ksmps* fits within the table length. For instance a table of length 16 will accommodate 1 to 16 samples, so this example will work with *ksmps* = 1 to 16.

Both these opcodes generate an error and deactivate the instrument if a table with length  $< ksmpls$  is selected. Likewise an error occurs if *kstart* is below 0 or greater than the highest entry in the table - if *kstart* = table length.

- *kstart* is intended to contain integer values between 0 and (table length - 1). Fractional values above this should not affect operation but do not achieve anything useful.
- These opcodes are not interpolating, and the *kstart* and *koff* parameters always have a range of 0 to (table length - 1) - not 0 to 1 as is available in other table read/write opcodes. *koff* can be outside this range but it is wrapped around by the final AND operation.
- These opcodes are permanently in wrap mode. When *koff* is 0, no wrapping needs to occur, since the *kstart++* index will always be within the table's normal range. *koff* not equal to 0 can lead to wrapping.
- The offset does not affect the number of read/write cycles performed, or the value written to *kstart* by *tablewa*.
- These opcodes cannot read or write the guardpoint. Use *tablegpw* to write the guardpoint after manipulations have been done with *tablewa*.

## Examples

```
kstart = 0

lab1:
  atemp tablewa ktabsource, kstart, 0 ; Read 5 values from table into an
    ; a-rate variable.

  atemp = log(atemp) ; Process the values using a-rate
    ; code.

  kstart tablewa ktabdest, atemp, 0 ; Write it back to the table

if ktemp 0 goto lab1 ; Loop until all table locations
  ; have been processed.
```

The above example shows a processing loop, which runs every k-cycle, reading each location in the table *ktabsource*, and writing the log of those values into the same locations of table *ktabdest*.

This enables whole tables, parts of tables (with offsets and different control loops) and data from several tables at once to be manipulated with a-rate code and written back to another (or to the same) table. This is a bit of a fudge, but it is faster than doing it with k-rate table read and write code.

Another application is:

```
kzero = 0
kloop = 0

kzero tablewa 23, asignal, 0 ; ksmpls a-rate samples written
  ; into locations 0 to (ksmps -1) of table 23.

lab1: ktemp tablewa kloop, 23 ; Start a loop which runs ksmpls times,
  ; in which each cycle processes one of
```

```
[ Some code to manipulate ] ; table 23's values with k-rate orchestra  
[ the value of ktemp. ] ; code.
```

```
tablew ktemp, kloop, 23 ; Write the processed value to the table.
```

```
kloop = kloop + 1 ; Increment the kloop, which is both the  
; pointer into the table and the loop  
if kloop < ksmps goto lab1 ; counter. Keep looping until all values  
; in the table have been processed.
```

```
asignal tablera 23, 0, 0 ; Copy the table contents back  
; to an a-rate variable.
```

*koff* -- This is an offset which is added to the sum of *kstart* and the internal index variable which steps through the table. The result is then ANDed with the lengthmask (000 0111 for a table of length 8 - or 9 with guardpoint) and that final index is used to read or write to the table. *koff* can be any value. It is converted into a long using the ANSI floor() function so that -4.3 becomes -5. This is what we would want when using offsets which range above and below zero.

Ideally this would be an optional variable, defaulting to 0, however with the existing Csound orchestra read code, such default parameters must be init time only. We want k-rate here, so we cannot have a default.

## See Also

*tablewa*

# tableseg

tableseg — Creates a new function table by making linear segments between values in stored function tables.

## Description

*tableseg* is like *linseg* but interpolate between values in a stored function tables. The result is a new function table passed internally to any following *vpvoc* which occurs before a subsequent *tableseg* (much like *lpread/lpreson* pairs work). The uses of these are described below under *vpvoc*.

## Syntax

```
tableseg ifn1, idur1, ifn2 [, idur2] [, ifn3] [...]
```

## Initialization

*ifn1*, *ifn2*, *ifn3*, etc. -- function table numbers. *ifn1*, *ifn2*, and so on, must be the same size.

*idur1*, *idur2*, etc. -- durations during which interpolation from one table to the next will take place.

## Examples

Here is an example of the *tableseg* opcode. It uses the file *tableseg.csd* [examples/tableseg.csd].

### Example 982. Example of the *tableseg* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o tableseg.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
; analyze "fox.wav" with PVANAL first
iend = p4
ktime line 0, p3, iend
tableseg p5, p3, p6 ;morph from table 1
asig vpvoc ktime, 1, "fox.pvx" ;to table 2
outs asig*3, asig*3

endin
</CsInstruments>
```

```
<CsScore>
f 1 0 512 9 .5 1 0
f 2 0 512 7 0 20 1 30 0 230 0 232 1

i 1 0 10 2.7 1 2
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvbufread, pvcross, pvinterp, pvread, tablexseg*

## Credits

Author: Richard Karpen  
Seattle, Wash  
1997

New in version 3.44

# tableshuffle

tableshuffle — shuffles the content of a function table so that each element of the source table is put into a different random position.

## Description

This opcode can be used in order to shuffle the content of function tables into a random order but without losing any of the elements. Imagine shuffling a deck of cards. Each element of the table is copied to a different random position. If that position was already chosen before then the next free position is chosen. The length of the table remains the same.

## Syntax

```
tableshuffle ktablenum
tableshufflei itablenum
```

## Performance

*ktablenum* or *itablenum* -- the number of the table to shuffle.

## Examples

Here is an example of the tableshuffle opcode. It uses the file *farey7shuffled.csd* [examples/farey7shuffled.csd].

### Example 983. Example of the tableshuffle opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-odac --rtaudio=alsa --midioutfile=farey7.mid
</CsOptions>
<CsInstruments>
sr=48000
ksmps=10
nchnls=1
0dbfs = 1

gidelta init 100
gimult init 101

;----- loop and trigger instrument 901 using a Farey Sequence polyrhythm
instr 1
kindx init 0
kindx2 init 0
ktrigger init 0
ktime_unit init p6
kstart init p4
kloop init p5
kinitndx init 0
kfn_times init gidelta
knote init 60
kbasenote init p8
```

```

ifundam init p7
ktrigger seqtime ktime_unit, kstart, kloop, kinitndx, kfn_times
if (ktrigger > 0 ) then
    kpitch = cpspch(ifundam)
    kmult tab kindx2, gimult
    kpitch = kpitch * kmult
    knote = kbasenote + kmult
    event "i", 901, 0, .4, .1, kpitch, kpitch * .9, .4, 5, .75, .8, 1.0, .15, .0, .125, .125, .25
    kindx = kindx + 1
    kindx = kindx % kloop
    kindx2 = kindx2 + 1
    kindx2 = kindx2 % kloop
    if (kindx2 == 0) then
        tableshuffle gimult
    endif
endif
endin ; 1

;----- basic 2 Operators FM algorithm -----
instr 901
inotedur = p3
imaxamp = p4 ;ampdb(p4)
icarrfreq = p5
imodfreq = p6
ilowndx = p7
indxdiff = p8-p7
knote = p27
aampenv linseg p9, p14*p3, p10, p15*p3, p11, p16*p3, p12, p17*p3, p13
adevenv linseg p18, p23*p3, p19, p24*p3, p20, p25*p3, p21, p26*p3, p22
amodosc oscili (ilowndx+indxdiff*adevenv)*imodfreq, imodfreq, 10
acarosc oscili imaxamp*aampenv, icarrfreq+amodosc, 10
out acarosc
;----- we also write down a midi track here -----
midion 1, knote, 100
endin ; 901

</CsInstruments>

<CsScore>
f10 0 4096 10 1
f100 0 -18 "farey" 7 1
f101 0 -18 "farey" 7 2

; p4 kstart := index offset into the Farey Sequence
; p5 kloop := end index into Farey Seq.
; p6 timefac := time in seconds for one loop to complete
; p7 fundam := fundamental of the FM instrument
; p8 basenote:= root pitch of the midi voice output
; note that pitch structures of the midi file output are not equivalent to the
; ones used for the FM real-time synthesis.

; start dur kstart kloop timefac fundam. basenote
i1 0.0 44 0 18 1 6.05 60
i1 4 40 0 18 3 7.05 72
i1 10 38 0 18 1.5 8 84
i1 15 50 0 18 1 5 48
i1 22 75 5 17 1.7 4 36
e
</CsScore>

</CsoundSynthesizer>

```

## See Also

*GEN farey, tablefilter, tablecopy*

## Credits

Author: Georg Boenn  
University of Glamorgan, UK

New in Csound version 5.13

# tablew

tablew — Change the contents of existing function tables.

## Description

This opcode operates on existing function tables, changing their contents. *tablew* is for writing at k- or at a-rates, with the table number being specified at init time. Using *tablew* with i-rate signal and index values is allowed, but the specified data will always be written to the function table at k-rate, not during the initialization pass. The valid combinations of variable types are shown by the first letter of the variable names.

## Syntax

```
tablew asig, andx, ifn [, ixmode] [, ixoff] [, iwgmode]
```

```
tablew isig, indx, ifn [, ixmode] [, ixoff] [, iwgmode]
```

```
tablew ksig, kndx, ifn [, ixmode] [, ixoff] [, iwgmode]
```

## Initialization

*asig, isig, ksig* -- The value to be written into the table.

*andx, indx, kndx* -- Index into table, either a positive number range matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* != 0)

*ifn* -- Table number. Must be >= 1. Floats are rounded down to an integer. If a table number does not point to a valid table, or the table has not yet been loaded (*GENOI*) then an error will result and the instrument will be de-activated.

*ixmode* (optional, default=0) -- index mode.

- 0 = *xndx* and *ixoff* ranges match the length of the table.
- !=0 = *xndx* and *ixoff* have a 0 to 1 range.

*ixoff* (optional, default=0) -- index offset.

- 0 = Total index is controlled directly by *xndx*, i.e. the indexing starts from the start of the table.
- !=0 = Start indexing from somewhere else in the table. Value must be positive and less than the table length (*ixmode* = 0) or less than 1 (*ixmode* != 0).

*iwgmode* (optional, default=0) -- Wrap and guardpoint mode.

- 0 = Limit mode.
- 1 = Wrap mode.
- 2 = Guardpoint mode.

## Performance

### Limit mode (0)

Limit the total index ( $ndx + ixoff$ ) to between 0 and the guard point. For a table of length 5, this means that locations 0 to 3 and location 4 (the guard point) can be written. A negative total index writes to location 0.

### Wrap mode (1)

Wrap total index value into locations 0 to E, where E is either one less than the table length or the factor of 2 number which is one less than the table length. For example, wrap into a 0 to 3 range - so that total index 6 writes to location 2.

### Guardpoint mode (2)

The guardpoint is written at the same time as location 0 is written - with the same value.

This facilitates writing to tables which are intended to be read with interpolation for producing smooth cyclic waveforms. In addition, before it is used, the total index is incremented by half the range between one location and the next, before being rounded down to the integer address of a table location.

Normally ( $igwmode = 0$  or  $1$ ) for a table of length 5 - which has locations 0 to 3 as the main table and location 4 as the guard point, a total index in the range of 0 to 0.999 will write to location 0. ("0.999" means just less than 1.0.) 1.0 to 1.999 will write to location 1 etc. A similar pattern holds for all total indexes 0 to 4.999 ( $igwmode = 0$ ) or to 3.999 ( $igwmode = 1$ ).  $igwmode = 0$  enables locations 0 to 4 to be written - with the guardpoint (4) being written with a potentially different value from location 0.

With a table of length 5 and the  $iwgmode = 2$ , then when the total index is in the range 0 to 0.499, it will write to locations 0 and 4. Range 0.5 to 1.499 will write to location 1 etc. 3.5 to 4.0 will also write to locations 0 and 4.

This way, the writing operation most closely approximates the results of interpolated reading. Guard point mode should only be used with tables that have a guardpoint.

Guardpoint mode is accomplished by adding 0.5 to the total index, rounding to the next lowest integer, wrapping it modulo the factor of two which is one less than the table length, writing the table (locations 0 to 3 in our example) and then writing to the guard point if index = 0.

*tablew* has no output value. The last three parameters are optional and have default values of 0.

## Caution with k-rate table numbers

At k-rate or a-rate, if a table number of  $< 1$  is given, or the table number points to a non-existent table, or to one which has a length of 0 (it is to be loaded from a file later) then an error will result and the instrument will be deactivated. *kfn* and *afn* must be initialized at the appropriate rate using *init*. Attempting to load an i-rate value into *kfn* or *afn* will result in an error.



### Warning

Note that *tablew* is always a k-rate opcode. This means that even its i-rate version runs at k-rate and will write the value of the i-rate variable. For this reason, the following code will not work as expected:

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
giFt ftgen 1, 0, 8, 2, 0
instr 1
indx = 0
    tablew 10, indx, giFt
ival tab_i indx, giFt
    print ival
endin
</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsoundSynthesizer>
```

Although it may seem this program should print a 10 to the console. It will print 0, because *tab\_i* will read the value at the initialization of the note, before the first performance pass, when *tablew* writes its value.

## See Also

*tableiw*, *tablewkt*

## Credits

Author: Robin Whittle  
Australia  
May 1997

# tablewa

tablewa — Writes tables in sequential locations.

## Description

This opcode writes to a table in sequential locations to and from an a-rate variable. Some thought is required before using it. It has at least two major, and quite different, applications which are discussed below.

## Syntax

```
kstart tablewa kfn, asig, koff
```

## Performance

*kstart* -- Where in table to read or write.

*kfn* -- i- or k-rate number of the table to read or write.

*asig* -- a-rate signal to read from when writing to the table.

*koff* -- i- or k-rate offset into table. Range unlimited - see explanation at end of this section.

In one application, it is intended to be used with one or with several *tablera* opcodes before a *tablewa* -- all sharing the same *kstart* variable.

These read from and write to sequential locations in a table at audio rates, with *ksmps* floats being written and read each cycle.

*tablera* starts reading from location *kstart*. *tablewa* starts writing to location *kstart*, and then writes to *kstart* with the number of the location one more than the one it last wrote. (Note that for *tablewa*, *kstart* is both an input and output variable.) If the writing index reaches the end of the table, then no further writing occurs and zero is written to *kstart*.

For instance, if the table's length was 16 (locations 0 to 15), and *ksmps* was 5. Then the following steps would occur with repetitive runs of the *tablewa* opcode, assuming that *kstart* started at 0.

Run Number	Initial kstart	Final kstart	Locations Written
1	0	5	0 1 2 3 4
2	5	10	5 6 7 8 9
3	10	15	10 11 12 13 14
4	15	0	15

This is to facilitate processing table data using standard a-rate orchestra code between the *tablera* and *tablewa* opcodes. They allow all Csound k-rate operators to be used (with caution) on a-rate variables - something that would only be possible otherwise by *ksmps* = 1, *downsamp* and *upsamp*.



### Several cautions

- The k-rate code in the processing loop is really running at a-rate, so time dependent functions like *port* and *oscil* work faster than normal - their code is expecting to be running at k-rate.

- This system will produce undesirable results unless the *ksmps* fits within the table length. For instance a table of length 16 will accommodate 1 to 16 samples, so this example will work with *ksmps* = 1 to 16.

Both these opcodes generate an error and deactivate the instrument if a table with length  $< ksmpls$  is selected. Likewise an error occurs if *kstart* is below 0 or greater than the highest entry in the table - if *kstart* = table length.

- *kstart* is intended to contain integer values between 0 and (table length - 1). Fractional values above this should not affect operation but do not achieve anything useful.
- These opcodes are not interpolating, and the *kstart* and *koff* parameters always have a range of 0 to (table length - 1) - not 0 to 1 as is available in other table read/write opcodes. *koff* can be outside this range but it is wrapped around by the final AND operation.
- These opcodes are permanently in wrap mode. When *koff* is 0, no wrapping needs to occur, since the *kstart++* index will always be within the table's normal range. *koff* not equal to 0 can lead to wrapping.
- The offset does not affect the number of read/write cycles performed, or the value written to *kstart* by *tablewa*.
- These opcodes cannot read or write the guardpoint. Use *tablegpw* to write the guardpoint after manipulations have been done with *tablewa*.

## Examples

```
kstart = 0

lab1:
  atemp tablewa ktabsource, kstart, 0 ; Read 5 values from table into an
    ; a-rate variable.

  atemp = log(atemp) ; Process the values using a-rate
    ; code.

  kstart tablewa ktabdest, atemp, 0 ; Write it back to the table

if ktemp 0 goto lab1 ; Loop until all table locations
  ; have been processed.
```

The above example shows a processing loop, which runs every k-cycle, reading each location in the table *ktabsource*, and writing the log of those values into the same locations of table *ktabdest*.

This enables whole tables, parts of tables (with offsets and different control loops) and data from several tables at once to be manipulated with a-rate code and written back to another (or to the same) table. This is a bit of a fudge, but it is faster than doing it with k-rate table read and write code.

Another application is:

```
kzero = 0
kloop = 0

kzero tablewa 23, asignal, 0 ; ksmpls a-rate samples written
  ; into locations 0 to (ksmps -1) of table 23.

lab1: ktemp tablewa kloop, 23 ; Start a loop which runs ksmpls times,
  ; in which each cycle processes one of
```

```
[ Some code to manipulate ] ; table 23's values with k-rate orchestra  
[ the value of ktemp. ] ; code.
```

```
tablew ktemp, kloop, 23 ; Write the processed value to the table.
```

```
kloop = kloop + 1 ; Increment the kloop, which is both the  
; pointer into the table and the loop  
if kloop < ksmps goto lab1 ; counter. Keep looping until all values  
; in the table have been processed.
```

```
asignal tablera 23, 0, 0 ; Copy the table contents back  
; to an a-rate variable.
```

*koff* -- This is an offset which is added to the sum of *kstart* and the internal index variable which steps through the table. The result is then ANDed with the lengthmask (000 0111 for a table of length 8 - or 9 with guardpoint) and that final index is used to read or write to the table. *koff* can be any value. It is converted into a long using the ANSI floor() function so that -4.3 becomes -5. This is what we would want when using offsets which range above and below zero.

Ideally this would be an optional variable, defaulting to 0, however with the existing Csound orchestra read code, such default parameters must be init time only. We want k-rate here, so we cannot have a default.

## Credits

Author: Robin Whittle  
Australia

# tablewkt

tablewkt — Change the contents of existing function tables.

## Description

This opcode operates on existing function tables, changing their contents. *tablewkt* uses a k-rate variable for selecting the table number. The valid combinations of variable types are shown by the first letter of the variable names.

## Syntax

```
tablewkt asig, andx, kfn [, ixmode] [, ixoff] [, iwgmode]
```

```
tablewkt ksig, kndx, kfn [, ixmode] [, ixoff] [, iwgmode]
```

## Initialization

*asig, ksig* -- The value to be written into the table.

*andx, kndx* -- Index into table, either a positive number range matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* != 0)

*kfn* -- Table number. Must be >= 1. Floats are rounded down to an integer. If a table number does not point to a valid table, or the table has not yet been loaded (*GEN01*) then an error will result and the instrument will be de-activated.

*ixmode* -- index mode. Default is zero.

- 0 = *xndx* and *ixoff* ranges match the length of the table.
- Not equal to 0 = *xndx* and *ixoff* have a 0 to 1 range.

*ixoff* -- index offset. Default is 0.

- 0 = Total index is controlled directly by *xndx*, i.e. the indexing starts from the start of the table.
- Not equal to 0 = Start indexing from somewhere else in the table. Value must be positive and less than the table length (*ixmode* = 0) or less than 1 (*ixmode* != 0).

*iwgmode* -- table writing mode. Default is 0.

- 0 = Limit mode.
- 1 = Wrap mode.
- 2 = Guardpoint mode.

## Performance

### Limit mode (0)

Limit the total index (*ndx* + *ixoff*) to between 0 and the guard point. For a table of length 5, this means that locations 0 to 3 and location 4 (the guard point) can be written. A negative total index writes to location 0.

## Wrap mode (1)

Wrap total index value into locations 0 to E, where E is one less than either the table length or the factor of 2 number which is one less than the table length. For example, wrap into a 0 to 3 range - so that total index 6 writes to location 2.

## Guardpoint mode (2)

The guardpoint is written at the same time as location 0 is written - with the same value.

This facilitates writing to tables which are intended to be read with interpolation for producing smooth cyclic waveforms. In addition, before it is used, the total index is incremented by half the range between one location and the next, before being rounded down to the integer address of a table location.

Normally (*igwmode* = 0 or 1) for a table of length 5 - which has locations 0 to 3 as the main table and location 4 as the guard point, a total index in the range of 0 to 0.999 will write to location 0. ("0.999" means just less than 1.0.) 1.0 to 1.999 will write to location 1 etc. A similar pattern holds for all total indexes 0 to 4.999 (*igwmode* = 0) or to 3.999 (*igwmode* = 1). *igwmode* = 0 enables locations 0 to 4 to be written - with the guardpoint (4) being written with a potentially different value from location 0.

With a table of length 5 and the *iwgmode* = 2, then when the total index is in the range 0 to 0.499, it will write to locations 0 and 4. Range 0.5 to 1.499 will write to location 1 etc. 3.5 to 4.0 will also write to locations 0 and 4.

This way, the writing operation most closely approximates the results of interpolated reading. Guard point mode should only be used with tables that have a guardpoint.

Guardpoint mode is accomplished by adding 0.5 to the total index, rounding to the next lowest integer, wrapping it modulo the factor of two which is one less than the table length, writing the table (locations 0 to 3 in our example) and then writing to the guard point if index = 0.

## Caution with k-rate table numbers

At k-rate or a-rate, if a table number of  $< 1$  is given, or the table number points to a non-existent table, or to one which has a length of 0 (it is to be loaded from a file later) then an error will result and the instrument will be deactivated. *kfn* and *afn* must be initialized at the appropriate rate using *init*. Attempting to load an i-rate value into *kfn* or *afn* will result in an error.

## See Also

*tableiw*, *tablew*

## Credits

Author: Robin Whittle  
Australia  
May 1997

# tablexkt

tablexkt — Reads function tables with linear, cubic, or sinc interpolation.

## Description

Reads function tables with linear, cubic, or sinc interpolation.

## Syntax

```
ares tablexkt xndx, kfn, kwarp, iwsiz [, ixmode] [, ixoff] [, iwrap]
```

## Initialization

*iwsiz* -- This parameter controls the type of interpolation to be used:

- 2: Use linear interpolation. This is the lowest quality, but also the fastest mode.
- 4: Cubic interpolation. Slightly better quality than *iwsiz* = 2, at the expense of being somewhat slower.
- 8 and above (up to 1024): sinc interpolation with window size set to *iwsiz* (should be an integer multiply of 4). Better quality than linear or cubic interpolation, but very slow. When transposing up, a *kwarp* value above 1 can be used for anti-aliasing (this is even slower).

*ixmode1* (optional) -- index data mode. The default value is 0.

- 0: raw index
- any non-zero value: normalized (0 to 1)



### Notes

if *tablexkt* is used to play back samples with looping (e.g. table index is generated by *lphasor*), there must be at least *iwsiz* / 2 extra samples after the loop end point for interpolation, otherwise audible clicking may occur (also, at least *iwsiz* / 2 samples should be before the loop start point).

*ixoff* (optional) -- amount by which index is to be offset. For a table with origin at center, use *tablesize* / 2 (raw) or 0.5 (normalized). The default value is 0.

*iwrap* (optional) -- wraparound index flag. The default value is 0.

- 0: Nowrap (index < 0 treated as index = 0; index >= *tablesize* (or 1.0 in normalized mode) sticks at the guard point).
- any non-zero value: Index is wrapped to the allowed range (not including the guard point in this case).



### Note

*iwrap* also applies to extra samples for interpolation.

## Performance

*ares* -- audio output

*xndx* -- table index

*kfn* -- function table number

*kwarp* -- if greater than 1, use  $\sin(x / \text{kwarp}) / x$  function for sinc interpolation, instead of the default  $\sin(x) / x$ . This is useful to avoid aliasing when transposing up (*kwarp* should be set to the transpose factor in this case, e.g. 2.0 for one octave), however it makes rendering up to twice as slow. Also, *iwsize* should be at least  $\text{kwarp} * 8$ . This feature is experimental, and may be optimized both in terms of speed and quality in new versions.



## Note

*kwarp* has no effect if it is less than, or equal to 1, or linear or cubic interpolation is used.

## Examples

Here is an example of the `tablexkt` opcode. It uses the file `tablexkt.csd` [examples/tablexkt.csd].

### Example 984. Example of the `tablexkt` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o tablexkt.wav -W ;; for file output any platform
</CsOptions>

<CsInstruments>
;Example by Jonathan Murphy

sr      = 44100
ksmps   = 10
nchnls  = 1

instr 1

ifn      = 1      ; query f1 as to number of samples
ilen     = nsamp(ifn)

itrns    = 4      ; transpose up 4 octaves
ilps     = 16     ; allow iwsize/2 samples at start
ilpe     = ilen - 16 ; and at end
imode    = 3      ; loop forwards and backwards
istrt    = 16     ; start 16 samples into loop

alphs    lphasor  itrns, ilps, ilpe, imode, istrt
; use lphasor as index
andx     = alphs

kfn      = 1      ; read f1
kwarp    = 4      ; anti-aliasing, should be same value as itrns above
iwsize   = 32     ; iwsize must be at least 8 * kwarp

atab     tablexkt andx, kfn, kwarp, iwsize

atab     = atab * 10000
```

```
        out      atab

    endin

</CsInstruments>

<CsScore>
f 1 0 262144 1 "beats.wav" 0 4 1
i1 0 60
e
</CsScore>

</CsoundSynthesizer>
```

## Credits

Author: Istvan Varga

January 2002

Example by: Jonathan Murphy 2006

New in version 4.18

# tablexseg

tablexseg — Creates a new function table by making exponential segments between values in stored function tables.

## Description

*tablexseg* is like *expseg* but interpolate between values in a stored function tables. The result is a new function table passed internally to any following *vpvoc* which occurs before a subsequent *tablexseg* (much like *lpread/lpreson* pairs work). The uses of these are described below under *vpvoc*.

## Syntax

```
tablexseg ifn1, idur1, ifn2 [, idur2] [, ifn3] [...]
```

## Initialization

*ifn1*, *ifn2*, *ifn3*, etc. -- function table numbers. *ifn1*, *ifn2*, and so on, must be the same size.

*idur1*, *idur2*, etc. -- durations during which interpolation from one table to the next will take place.

## Examples

Here is an example of the *tablexseg* opcode. It uses the file *tablexseg.csd* [examples/tablexseg.csd].

### Example 985. Example of the *tablexseg* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o tablexseg.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
; analyze "fox.wav" with PVANAL first
iend = p4
ktime line 0, p3, iend
tablexseg p5, p3, p6 ;morph from table 1
asig vpvoc ktime, 1, "fox.pvx" ;to table 2
outs asig*3, asig*3

endin
</CsInstruments>
```

```
<CsScore>
f 1 0 512 9 .5 1 0
f 2 0 512 5 1 60 0.01 390 0.01 62 1

i 1 0 5 2.7 1 2
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvbufread, pvcross, pvinterp, pvread, tableseg*

## Credits

Author: Richard Karpen  
Seattle, WA USA  
1997

# tabmorph

tabmorph — Allow morphing between a set of tables.

## Description

*tabmorph* allows morphing between a set of tables of the same size, by means of a weighted average between two currently selected tables.

## Syntax

```
kout tabmorph kindex, kweightpoint, ktabnum1, ktabnum2, \  
ifn1, ifn2 [, ifn3, ifn4, ...,ifnN]
```

## Initialization

*ifn1, ifn2 [, ifn3, ifn4, ..., ifnN]* - function table numbers. This is a set of chosen tables the user want to use in the morphing. All tables must have the same length. Be aware that only two of these tables can be chosen for the morphing at one time. Since it is possible to use non-integer numbers for the *ktabnum1* and *ktabnum2* arguments, the morphing is the result from the interpolation between adjacent consecutive tables of the set.

## Performance

*kout* - The output value for index *kindex*, resulting from morphing two tables (see below).

*kindex* - main index of the morphed resultant table. The range is 0 to *table\_length* (not included).

*kweightpoint* - the weight of the influence of a pair of selected tables in the morphing. The range of this argument is 0 to 1. A zero makes it output the first table unaltered, a 1 makes it output the second table of the pair unaltered. All intermediate values between 0 and 1 determine the gradual morphing between the two tables of the pair.

*ktabnum1* - the first table chosen for the morphing. This number doesn't express the table number directly, but the position of the table in the set sequence (starting from 0 to N-1). If this number is an integer, the corresponding table will be chosen unaltered. If it contains fractional values, then an interpolation with the next adjacent table will result.

*ktabnum2* - the second table chosen for the morphing. This number doesn't express the table number directly, but the position of the table in the set sequence (starting from 0 to N-1). If this number is an integer, corresponding table will be chosen unaltered. If it contains fractional values, then an interpolation with the next adjacent table will result.

The *tabmorph* family of opcodes is similar to the *table* family, but allows morphing between two tables chosen into a set of tables. Firstly the user has to provide a set of tables of equal length (*ifn2 [, ifn3, ifn4, ..., ifnN]*). Then he can choose a pair of tables in the set in order to perform the morphing: *ktabnum1* and *ktabnum2* are filled with numbers (zero represents the first table in the set, 1 the second, 2 the third and so on). Then determine the morphing between the two chosen tables with the *kweightpoint* parameter. After that the resulting table can be indexed with the *kindex* parameter like a normal *table* opcode. If the value of this parameter surpasses the length of tables (which must be the same for all tables), then it is wrapped around.

*tabmorph* acts similarly to the *table* opcode, that is, without using interpolation. This means that it truncates the fractional part of the *kindex* argument. Anyway, fractional parts of *ktabnum1* and *ktabnum2* are significant, resulting in linear interpolation between the same element of two adjacent subsequent tables.

## Examples

Here is an example of the *tabmorph* opcode. It uses the file *tabmorph.csd* [examples/tabmorph.csd].

### Example 986. Example of the *tabmorph* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o tabmorph.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine  ftgen 0, 0, 8193, 10, 1 ;sine wave
giSquare ftgen 0, 0, 8193, 7, 1, 4096, 1, 0, -1, 4096, -1 ;square wave
giTri   ftgen 0, 0, 8193, 7, 0, 2048, 1, 4096, -1, 2048, 0 ;triangle wave
giSaw   ftgen 0, 0, 8193, 7, 1, 8192, -1 ;sawtooth wave, downward slope

instr 1

iamp = .7
kindex phasor 440 ;read table value at this index
kindex = kindex*8192 ;for all 8192 index points
kweightpoint = 0.5 ;set weightpoint
ktabnum1 line 0, p3, 3 ;morph through all tables
ktabnum2 = 2 ;set to triangle wave
ksig tabmorph kindex, kweightpoint, ktabnum1, ktabnum2, giSine, giSquare, giTri, giSaw
ksig = ksig*iamp
asig interp ksig ;convert to audio signal
outs asig, asig

endin
</CsInstruments>
<CsScore>

i1 0 5
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*table*, *tabmorphi*, *tabmorpha*, *tabmorphak*, *ftmorf*,

## Credits

Author: Gabriel Maldonado

New in version 5.06

# tabmorpha

tabmorpha — Allow morphing between a set of tables at audio rate with interpolation.

## Description

*tabmorpha* allows morphing between a set of tables of the same size, by means of a weighted average between two currently selected tables.

## Syntax

```
aout tabmorpha aindex, aweightpoint, atabnum1, atabnum2, \  
ifn1, ifn2 [, ifn3, ifn4, ... ifnN]
```

## Initialization

*ifn1, ifn2, ifn3, ifn4, ... ifnN* - function table numbers. This is a set of chosen tables the user want to use in the morphing. All tables must have the same length. Be aware that only two of these tables can be chosen for the morphing at one time. Since it is possible to use non-integer numbers for the *atabnum1* and *atabnum2* arguments, the morphing is the result from the interpolation between adjacent consecutive tables of the set.

## Performance

*aout* - The output value for index *aindex*, resulting from morphing two tables (see below).

*aindex* - main index index of the morphed resultant table. The range is 0 to *table\_length* (not included).

*aweightpoint* - the weight of the influence of a pair of selected tables in the morphing. The range of this argument is 0 to 1. A zero makes it output the first table unaltered, a 1 makes it output the second table of the pair unaltered. All intermediate values between 0 and 1 determine the gradual morphing between the two tables of the pair.

*atabnum1* - the first table chosen for the morphing. This number doesn't express the table number directly, but the position of the table in the set sequence (starting from 0 to N-1). If this number is an integer, the corresponding table will be chosen unaltered. If it contains fractional values, then an interpolation with the next adjacent table will result.

*atabnum2* - the second table chosen for the morphing. This number doesn't express the table number directly, but the position of the table in the set sequence (starting from 0 to N-1). If this number is an integer, corresponding table will be chosen unaltered. If it contains fractional values, then an interpolation with the next adjacent table will result.

The *tabmorpha* family of opcodes is similar to the *table* family, but allows morphing between two tables chosen into a set of tables. Firstly the user has to provide a set of tables of equal length (*ifn2* [, *ifn3*, *ifn4*, ...*ifnN*]). Then he can choose a pair of tables in the set in order to perform the morphing: *atabnum1* and *atabnum2* are filled with numbers (zero represents the first table in the set, 1 the second, 2 the third and so on). Then determine the morphing between the two chosen tables with the *aweightpoint* parameter. After that the resulting table can be indexed with the *aindex* parameter like a normal *table* opcode. If the value of this parameter surpasses the length of tables (which must be the same for all tables), then it is wrapped around.

*tabmorpha* is the audio-rate version of *tabmorphi* (it uses interpolation). All input arguments work at a-rate.

## Examples

Here is an example of the *tabmorpha* opcode. It uses the file *tabmorpha.csd* [examples/tabmorpha.csd].

### Example 987. Example of the *tabmorpha* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o tabmorpha.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine ftgen 0, 0, 8193, 10, 1      ; sine wave
giSquare ftgen 0, 0, 8193, 7, 1, 4096, 1, 0, -1, 4096, -1 ; square wave
giTri ftgen 0, 0, 8193, 7, 0, 2048, 1, 4096, -1, 2048, 0 ; triangle wave
giSaw ftgen 0, 0, 8193, 7, 1, 8192, -1 ; sawtooth wave, downward slope

instr 1

iamp = .7
aindex phasor 110 ; read table value at this index
aweightpoint = 0 ; set weightpoint
atabnum1 line 0, p3, 3 ; morph through all tables
atabnum2 = 2 ; set to triangle wave
asig tabmorpha aindex, aweightpoint, atabnum1,atabnum2, giSine, giSquare, giTri, giSaw
asig = asig*iamp
outs asig, asig

endin
</CsInstruments>
<CsScore>

i1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*table*, *tabmorph*, *tabmorphi*, *tabmorphak*, *fmorf*,

## Credits

Author: Gabriel Maldonado

New in version 5.06

# tabmorphak

tabmorphak — Allow morphing between a set of tables at audio rate with interpolation.

## Description

*tabmorphak* allows morphing between a set of tables of the same size, by means of a weighted average between two currently selected tables.

## Syntax

```
aout tabmorphak aindex, kweightpoint, ktabnum1, ktabnum2, \  
ifn1, ifn2 [, ifn3, ifn4, ... ifnN]
```

## Initialization

*ifn1, ifn2, ifn3, ifn4, ... ifnN* - function table numbers. This is a set of chosen tables the user want to use in the morphing. All tables must have the same length. Be aware that only two of these tables can be chosen for the morphing at one time. Since it is possible to use non-integer numbers for the *ktabnum1* and *ktabnum2* arguments, the morphing is the result from the interpolation between adjacent consecutive tables of the set.

## Performance

*aout* - The output value for index *aindex*, resulting from morphing two tables (see below).

*aindex* - main index index of the morphed resultant table. The range is 0 to *table\_length* (not included).

*kweightpoint* - the weight of the influence of a pair of selected tables in the morphing. The range of this argument is 0 to 1. A zero makes it output the first table unaltered, a 1 makes it output the second table of the pair unaltered. All intermediate values between 0 and 1 determine the gradual morphing between the two tables of the pair.

*ktabnum1* - the first table chosen for the morphing. This number doesn't express the table number directly, but the position of the table in the set sequence (starting from 0 to N-1). If this number is an integer, the corresponding table will be chosen unaltered. If it contains fractional values, then an interpolation with the next adjacent table will result.

*ktabnum2* - the second table chosen for the morphing. This number doesn't express the table number directly, but the position of the table in the set sequence (starting from 0 to N-1). If this number is an integer, corresponding table will be chosen unaltered. If it contains fractional values, then an interpolation with the next adjacent table will result.

The *tabmorphak* family of opcodes is similar to the *table* family, but allows morphing between two tables chosen into a set of tables. Firstly the user has to provide a set of tables of equal length (*ifn2 [, ifn3, ifn4, ... ifnN]*). Then he can choose a pair of tables in the set in order to perform the morphing: *ktabnum1* and *ktabnum2* are filled with numbers (zero represents the first table in the set, 1 the second, 2 the third and so on). Then determine the morphing between the two chosen tables with the *kweightpoint* parameter. After that the resulting table can be indexed with the *aindex* parameter like a normal table opcode. If the value of this parameter surpasses the length of tables (which must be the same for all tables), then it is wrapped around.

*tabmorphak* works at a-rate, but *kweightpoint*, *ktabnum1* and *ktabnum2* are working at k-rate, making it more efficient than *tabmorpha*, since there are less calculations. Except the rate of these three arguments, it is identical to *tabmorpha*.

## Examples

Here is an example of the *tabmorphak* opcode. It uses the file *tabmorphak.csd* [examples/tabmorphak.csd].

### Example 988. Example of the *tabmorphak* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o tabmorphak.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine  ftgen 0, 0, 8193, 10, 1      ; sine wave
giSquare ftgen 0, 0, 8193, 7, 1, 4096, 1, 0, -1, 4096, -1 ; square wave
giTri    ftgen 0, 0, 8193, 7, 0, 2048, 1, 4096, -1, 2048, 0 ; triangle wave
giSaw    ftgen 0, 0, 8193, 7, 1, 8192, -1      ; sawtooth wave, downward slope

instr 1

iamp = .7
aindex phasor 110 ; read table value at this index
kweightpoint expon 0.001, p3, 1 ; using the weightpoint to morph between two tables exponentially
ktabnum1 = p4 ; first wave, it morphs to
ktabnum2 = p5 ; the second wave
asig tabmorphak aindex, kweightpoint, ktabnum1,ktabnum2, giSine, giSquare, giTri, giSaw
asig = asig*iamp
outs asig, asig

endin
</CsInstruments>
<CsScore>

i1 0 5 0 1 ;from sine to square wave
i1 6 5 2 3 ;from triangle to saw
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*table*, *tabmorph*, *tabmorphi*, *tabmorpha*, *fmorf*,

## Credits

Author: Gabriel Maldonado

New in version 5.06

# tabmorphi

tabmorphi — Allow morphing between a set of tables with interpolation.

## Description

*tabmorphi* allows morphing between a set of tables of the same size, by means of a weighted average between two currently selected tables.

## Syntax

```
kout tabmorphi kindex, kweightpoint, ktabnum1, ktabnum2, \  
ifn1, ifn2 [, ifn3, ifn4, ..., ifnN]
```

## Initialization

*ifn1, ifn2 [, ifn3, ifn4, ..., ifnN]* - function table numbers. This is a set of chosen tables the user want to use in the morphing. All tables must have the same length. Be aware that only two of these tables can be chosen for the morphing at one time. Since it is possible to use non-integer numbers for the *ktabnum1* and *ktabnum2* arguments, the morphing is the result from the interpolation between adjacent consecutive tables of the set.

## Performance

*kout* - The output value for index *kindex*, resulting from morphing two tables (see below).

*kindex* - main index index of the morphed resultant table. The range is 0 to *table\_length* (not included).

*kweightpoint* - the weight of the influence of a pair of selected tables in the morphing. The range of this argument is 0 to 1. A zero makes it output the first table unaltered, a 1 makes it output the second table of the pair unaltered. All intermediate values between 0 and 1 determine the gradual morphing between the two tables of the pair.

*ktabnum1* - the first table chosen for the morphing. This number doesn't express the table number directly, but the position of the table in the set sequence (starting from 0 to N-1). If this number is an integer, the corresponding table will be chosen unaltered. If it contains fractional values, then an interpolation with the next adjacent table will result.

*ktabnum2* - the second table chosen for the morphing. This number doesn't express the table number directly, but the position of the table in the set sequence (starting from 0 to N-1). If this number is an integer, corresponding table will be chosen unaltered. If it contains fractional values, then an interpolation with the next adjacent table will result.

The *tabmorphi* family of opcodes is similar to the *table* family, but allows morphing between two tables chosen into a set of tables. Firstly the user has to provide a set of tables of equal length (*ifn2 [, ifn3, ifn4, ..., ifnN]*). Then he can choose a pair of tables in the set in order to perform the morphing: *ktabnum1* and *ktabnum2* are filled with numbers (zero represents the first table in the set, 1 the second, 2 the third and so on). Then determine the morphing between the two chosen tables with the *kweightpoint* parameter. After that the resulting table can be indexed with the *kindex* parameter like a normal *table* opcode. If the value of this parameter surpasses the length of tables (which must be the same for all tables), then it is wrapped around.

*tabmorphi* is identical to *tabmorph*, but it performs linear interpolation for non-integer values of *kindex*, much like *tablei*.

## Examples

Here is an example of the *tabmorphi* opcode. It uses the file *tabmorphi.csd* [examples/tabmorphi.csd].

### Example 989. Example of the *tabmorphi* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o tabmorphi.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine   ftgen 0, 0, 8193, 10, 1      ;sine wave
giSquare ftgen 0, 0, 8193, 7, 1, 4096, 1, 0, -1, 4096, -1 ;square wave
giTri    ftgen 0, 0, 8193, 7, 0, 2048, 1, 4096, -1, 2048, 0 ;triangle wave
giSaw    ftgen 0, 0, 8193, 7, 1, 8192, -1      ;sawtooth wave, downward slope

instr    1

iamp     = .7
kindex  phasor 440      ;read table value at this index
kindex  = kindex*8192   ;for all 8192 index points
kweightpoint = 0.5     ;set weightpoint
ktabnum1 line 0, p3, 3  ;morph through all tables
ktabnum2 = 2           ;set to triangle wave
ksig    tabmorphi kindex, kweightpoint, ktabnum1, ktabnum2, giSine, giSquare, giTri, giSaw
ksig    = ksig*iamp
asig    interp ksig     ;convert to audio signal
outs    asig, asig

endin
</CsInstruments>
<CsScore>

i1 0 5
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*table*, *tabmorph*, *tabmorpha*, *tabmorphak*, *fmorf*,

## Credits

Author: Gabriel Maldonado

New in version 5.06

# tabplay

tabplay — Playing-back control signals.

## Description

Plays-back control-rate signals on trigger-temporization basis.

## Syntax

```
tabplay ktrig, knumtics, kfn, kout1 [,kout2,..., koutN]
```

## Performance

*ktrig* -- starts playing when non-zero.

*knumtics* -- stop recording or reset playing pointer to zero when the number of tics defined by this argument is reached.

*kfn* -- table where k-rate signals are recorded.

*kout1,...,koutN* -- playback output signals.

The *tabplay* and *tabrec* opcodes allow to record/playback control signals on trigger-temporization basis.

*tabplay* plays back a group of k-rate signals, previously recorded by *tabrec* into a table. Each time *ktrig* argument is triggered, an internal counter is increased of one unit. After *knumtics* trigger impluses are received by *ktrig* argument, the internal counter is zeroed and playback is restarted from the beginning, in looping style.

These opcodes can be used like a sort of "middle-term" memory that "remembers" generated signals. Such memory can be used to supply generative music with a coherent iterative compositional structure.

## See Also

*tabrec*

## Credits

Written by Gabriel Maldonado.

# tabrec

tabrec — Recording of control signals.

## Description

Records control-rate signals on trigger-temporization basis.

## Syntax

```
tabrec  ktrig_start, ktrig_stop, knumtics, kfn, kin1 [,kin2,...,kinN]
```

## Performance

*ktrig\_start* -- start recording when non-zero.

*ktrig\_stop* -- stop recording when *knumtics* trigger impulses are received by this input argument.

*knumtics* -- stop recording or reset playing pointer to zero when the number of tics defined by this argument is reached.

*kfn* -- table where k-rate signals are recorded.

*kin1*,...,*kinN* -- input signals to record.

The *tabrec* and *tabplay* opcodes allow to record/playback control signals on trigger-temporization basis.

*tabrec* opcode records a group of k-rate signals by storing them into *kfn* table. Each time *ktrig\_start* is triggered, *tabrec* resets the table pointer to zero and begins to record. Recording phase stops after *knumtics* trigger impulses have been received by *ktrig\_stop* argument.

These opcodes can be used like a sort of "middle-term" memory that "remembers" generated signals. Such memory can be used to supply generative music with a coherent iterative compositional structure.

## See Also

*tabplay*

## Credits

Written by Gabriel Maldonado.

# tabsum

tabsum — Adding values in a range of a table.

## Description

Sums the values in an f-table in a consecutive range.

## Syntax

```
kr tabsum ifn[[, kmin] [, kmax]]
```

## Initialization

*ifn* -- table number

## Performance

*kr* -- input signal to write.

*kmin*, *kmax* -- range of the table to sum. If omitted or zero they default to 0 to length of the table.

## Examples

Here is an example of the tabsum opcode. It uses the file *tabsum.csd* [examples/tabsum.csd].

### Example 990. Example of the tabsum opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
; For Non-realtime ouput leave only the line below:
; -o tab.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 2205
nchnls = 1
0dbfs = 1

instr 1      ;; Give a value to the increment
  kmax = 256
  knorm tabsum 1, 0, kmax
  gkinc = knorm/10
endin

instr 2
  kmax = 256
  kx = rnd(kmax)
```

```
krnd  tabsum 1, 0, kx
knorm tabsum 1, 0, kmax
kvar  = krnd / knorm      ;;; now n [0,1] range
asig  oscil  kvar, p4, 2
      out   asig
;;; Make randomness give 1 more often
kc    tab    0, 1
      tablew kc+gkinc, 0, 1
endin
</CsInstruments>

<CsScore>
f1 0 256 21 1
f2 0 4096 10 1
i1 0 0.1
i2 0.1 3 440
e

</CsScore>

</CsoundSynthesizer>
```

## See Also

Vectorial opcodes

## Credits

Author: John ffitch  
Codemist Ltd  
2009

New in version 5.11

# tab2pvs

tab2pvs — Copies spectral data from k-rate arrays (or t-variables.). Also known as pvsfromarray.

## Description

Copies a pvs frame from a t-variable or k-rate array. Currently only AMP+FREQ is produced. This opcode requires the t-type to be defined, which means it only works in the new bison/flex-based parser.

## Syntax

```
fsig tab2pvs tvar|karr[[],ihopsize, iwinsize, iwintype]
fsig tab2pvs kmags[], kfreqs[[],ihopsize, iwinsize, iwintype]
```

## Performance

*tvar* -- k-rate array (or t-variable) containing the input. It is produced at every k-period, but may not contain a new frame, pvs frames are produced at their own frame rate that is independent of kr. The size of this vector will determine the fftsize,  $N = \text{size} - 2$ .

*kmags[], kfreqs[]* -- k-rate arrays containing the input magnitude and frequency values. The output is produced at every k-period, but may not contain a new frame, pvs frames are produced at their own frame rate that is independent of kr. These vectors must be the same size and they will determine the fftsize,  $N = \text{size} - 2$ .

*fsig* -- output fsig to be copied.

*iolap* -- size of the analysis overlap, defaults to *isize*/4.

*iwinsize* -- size of the analysis window, defaults to *isize*.

*iwintype* -- type of analysis window, defaults to 1, Hanning.

## Examples

### Example 991. Example

```
karr[] init 1026
...
fsigl tab2pvs karr
aout pvsynth fsigl
```

## Credits

Author: Victor Lazzarini  
October 2011

New plugin in version 5

October 2011.

# tambourine

tambourine — Semi-physical model of a tambourine sound.

## Description

*tambourine* is a semi-physical model of a tambourine sound. It is one of the PhISEM percussion opcodes. PhISEM (Physically Informed Stochastic Event Modeling) is an algorithmic approach for simulating collisions of multiple independent sound producing objects.

## Syntax

```
ares tambourine kamp, idettack [, inum] [, idamp] [, imaxshake] [, ifreq] \  
    [, ifreq1] [, ifreq2]
```

## Initialization

*idettack* -- period of time over which all sound is stopped

*inum* (optional) -- The number of beads, teeth, bells, timbrels, etc. If zero, the default value is 32.

*idamp* (optional) -- the damping factor, as part of this equation:

$$\text{damping\_amount} = 0.9985 + (\text{idamp} * 0.002)$$

The default *damping\_amount* is 0.9985 which means that the default value of *idamp* is 0. The maximum *damping\_amount* is 1.0 (no damping). This means the maximum value for *idamp* is 0.75.

The recommended range for *idamp* is usually below 75% of the maximum value.

*imaxshake* (optional, default=0) -- amount of energy to add back into the system. The value should be in range 0 to 1.

*ifreq* (optional) -- the main resonant frequency. The default value is 2300.

*ifreq1* (optional) -- the first resonant frequency. The default value is 5600.

*ifreq2* (optional) -- the second resonant frequency. The default value is 8100.

## Performance

*kamp* -- Amplitude of output. Note: As these instruments are stochastic, this is only an approximation.

## Examples

Here is an example of the tambourine opcode. It uses the file *tambourine.csd* [examples/tambourine.csd].

### Example 992. Example of the tambourine opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o tambourine.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

idamp = p4
asig  tambourine .8, 0.01, 30, idamp, 0.4
outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 .2 0
i 1 + .2 >
i 1 + 1 .7
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*bamboo, dripwater, guiro, sleighbells*

## Credits

Author: Perry Cook, part of the PhISEM (Physically Informed Stochastic Event Modeling)  
Adapted by John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 4.07

Added notes by Rasmus Ekman on May 2002.

# tan

tan — Performs a tangent function.

## Description

Returns the tangent of  $x$  ( $x$  in radians).

## Syntax

```
tan(x) (no rate  
restriction)  
tan(k/i[]) (k- or i-arrays )
```

## Examples

Here is an example of the tan opcode. It uses the file *tan.csd* [examples/tan.csd].

### Example 993. Example of the tan opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
; Audio out  Audio in  
-odac          -iadc      ;;RT audio I/O  
; For Non-realtime ouput leave only the line below:  
; -o tan.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>  
  
; Initialize the global variables.  
sr = 44100  
kr = 4410  
ksmps = 10  
nchnls = 1  
  
; Instrument #1.  
instr 1  
  irad = 25  
  i1 = tan(irad)  
  
  print i1  
endin  
  
</CsInstruments>  
<CsScore>  
  
; Play Instrument #1 for one second.  
i 1 0 1  
e  
  
</CsScore>
```

```
</CsoundSynthesizer>
```

Its output should include a line like this:

```
instr 1: i1 = -0.134
```

## See Also

*cos, cosh, cosinv, sin, sinh, sininv, tan, taninv*

## Credits

Written by John ffitch.

New in version 3.47

Example written by Kevin Conder.

# tanh

tanh — Performs a hyperbolic tangent function.

## Description

Returns the hyperbolic tangent of  $x$ .

## Syntax

```
tanh(x) (no rate  
restriction)  
tanh(k/i[]) (k- or i-arrays )
```

## Examples

Here is an example of the tanh opcode. It uses the file *tanh.csd* [examples/tanh.csd].

### Example 994. Example of the tanh opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
; Audio out  Audio in  
-odac          -iadc      ;;RT audio I/O  
; For Non-realtime ouput leave only the line below:  
; -o tanh.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>  
  
; Initialize the global variables.  
sr = 44100  
kr = 4410  
ksmps = 10  
nchnls = 1  
0dbfs = 1  
  
; Instrument #1.  
instr 1  
  asig1 vco 1, 440, 2, 0.4, 1  
  asig2 vco 1, 800, 3, 0.5, 1  
  asig = asig1+asig2      ;; will go out of range  
  out tanh(asig)        ;; but tanh is a limiter  
endin  
  
</CsInstruments>  
<CsScore>  
  
f1 0 65536 10 1  
; Play Instrument #1 for one second.  
i 1 0 1  
e
```

```
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*cos, cosh, cosinv, sin, sinh, sininv, tan, taninv*

## Credits

Author: John ffitch

New in version 3.47

# taninv

taninv — Performs an arctangent function.

## Description

Returns the arctangent of  $x$  ( $x$  in radians).

## Syntax

```
taninv(x) (no rate  
restriction)  
taninv(k/i[]) (k- or i-arrays )
```

## Examples

Here is an example of the taninv opcode. It uses the file *taninv.csd* [examples/taninv.csd].

### Example 995. Example of the taninv opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
; Audio out  Audio in  
-odac          -iadc      ;;RT audio I/O  
; For Non-realtime ouput leave only the line below:  
; -o taninv.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>  
  
; Initialize the global variables.  
sr = 44100  
kr = 4410  
ksmps = 10  
nchnls = 1  
  
; Instrument #1.  
instr 1  
  irad = 0.5  
  i1 = taninv(irad)  
  
  print i1  
endin  
  
</CsInstruments>  
<CsScore>  
  
; Play Instrument #1 for one second.  
i 1 0 1  
e  
  
</CsScore>
```

```
</CsoundSynthesizer>
```

Its output should include a line like this:

```
instr 1: i1 = 0.464
```

## See Also

*cos, cosh, cosinv, sin, sinh, sininv, tan, tanh, taninv2*

## Credits

Author: John ffitch

New in version 3.48

Example written by Kevin Conder.

# taninv2

taninv2 — Returns an arctangent.

## Description

Returns the arctangent of  $iy/ix$ ,  $ky/kx$ , or  $ay/ax$ .

## Syntax

```
ares taninv2 ay, ax
ires taninv2 iy, ix
kres taninv2 ky, kx
...taninv2(ky, kx)... (no rate restriction)
```

Returns the arctangent of  $iy/ix$ ,  $ky/kx$ , or  $ay/ax$ . If  $y$  is zero, *taninv2* returns zero regardless of the value of  $x$ . If  $x$  is zero, the return value is:

- $\pi/2$ , if  $y$  is positive.
- $-\pi/2$ , if  $y$  is negative.
- $0$ , if  $y$  is  $0$ .

## Initialization

*iy*, *ix* -- values to be converted

## Performance

*ky*, *kx* -- control rate signals to be converted

*ay*, *ax* -- audio rate signals to be converted

## Examples

Here is an example of the *taninv2* opcode. It uses the file *taninv2.csd* [examples/taninv2.csd].

### Example 996. Example of the *taninv2* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o taninv2.wav -W ;; for file output any platform
</CsOptions>
```

```
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Returns the arctangent for 1/2.
i1 taninv2 1, 2

print i1
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
e

</CsScore>
</CsSoundSynthesizer>
```

Its output should include a line like this:

```
instr 1: i1 = 0.464
```

## See Also

*taninv*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
April 1998

Example written by Kevin Conder.

New in Csound version 3.48

Corrected on May 2002, thanks to Istvan Varga.

Available as a function in version 6.00

# tbvcf

tbvcf — Models some of the filter characteristics of a Roland TB303 voltage-controlled filter.

## Description

This opcode attempts to model some of the filter characteristics of a Roland TB303 voltage-controlled filter. Euler's method is used to approximate the system, rather than traditional filter methods. Cutoff frequency, Q, and distortion are all coupled. Empirical methods were used to try to unentwine, but frequency is only approximate as a result. Future fixes for some problems with this opcode may break existing orchestras relying on this version of *tbvcf*.

## Syntax

```
ares tbvcf asig, xfco, xres, kdist, kasym [, iskip]
```

## Initialization

*iskip* (optional, default=0) -- if non zero skip the initialisation of the filter. (New in Csound version 4.23f13 and 5.0)

## Performance

*asig* -- input signal. Should be normalized to  $\pm 1$ .

*xfco* -- filter cutoff frequency. Optimum range is 10,000 to 1500. Values below 1000 may cause problems.

*xres* -- resonance or Q. Typically in the range 0 to 2.

*kdist* -- amount of distortion. Typical value is 2. Changing *kdist* significantly from 2 may cause odd interaction with *xfco* and *xres*.

*kasym* -- asymmetry of resonance. Typically in the range 0 to 1.

## Examples

Here is an example of the *tbvcf* opcode. It uses the file *tbvcf.csd* [examples/tbvcf.csd].

### Example 997. Example of the *tbvcf* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o tbvcf.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

;-----
; TBVCF Test
```

## Orchestra Opcodes and Operators

---

```

; Coded by Hans Mikelson December, 2000
;-----
sr = 44100 ; Sample rate
ksmps = 10 ; Samples/Kontrol period
nchnls = 2 ; Normal stereo
0dbfs = 1

instr 10

idur = p3 ; Duration
iamp = p4 ; Amplitude
ifqc = cpspch(p5) ; Pitch to frequency
ipanl = sqrt(p6) ; Pan left
ipanr = sqrt(1-p6) ; Pan right
iq = p7
idist = p8
iasym = p9

kdclck linseg 0, .002, 1, idur-.004, 1, .002, 0 ; Declick envelope
kfco expseg 10000, idur, 1000 ; Frequency envelope
ax vco 1, ifqc, 2, 0.5 ; Square wave
ay tbvcf ax, kfco, iq, idist, iasym ; TB-VCF
ay buthp ay/1, 100 ; Hi-pass

outs ay*iamp*ipanl*kdclck, ay*iamp*ipanr*kdclck
endin

</CsInstruments>
<CsScore>

f1 0 65536 10 1

; TeeBee Test
; Sta Dur Amp Pitch Pan Q Dist1 Asym
i10 0 0.2 0.5 7.00 .5 0.0 2.0 0.0
i10 0.3 0.2 0.5 7.00 .5 0.8 2.0 0.0
i10 0.6 0.2 0.5 7.00 .5 1.6 2.0 0.0
i10 0.9 0.2 0.5 7.00 .5 2.4 2.0 0.0
i10 1.2 0.2 0.5 7.00 .5 3.2 2.0 0.0
; i10 1.5 0.2 0.5 7.00 .5 4.0 2.0 0.0
i10 1.8 0.2 0.5 7.00 .5 0.0 2.0 0.25
i10 2.1 0.2 0.5 7.00 .5 0.8 2.0 0.25
i10 2.4 0.2 0.5 7.00 .5 1.6 2.0 0.25
i10 2.7 0.2 0.5 7.00 .5 2.4 2.0 0.25
i10 3.0 0.2 0.5 7.00 .5 3.2 2.0 0.25
i10 3.3 0.2 0.5 7.00 .5 4.0 2.0 0.25
i10 3.6 0.2 0.5 7.00 .5 0.0 2.0 0.5
i10 3.9 0.2 0.5 7.00 .5 0.8 2.0 0.5
i10 4.2 0.2 0.5 7.00 .5 1.6 2.0 0.5
i10 4.5 0.2 0.5 7.00 .5 2.4 2.0 0.5
i10 4.8 0.2 0.5 7.00 .5 3.2 2.0 0.5
i10 5.1 0.2 0.5 7.00 .5 4.0 2.0 0.5
i10 5.4 0.2 0.5 7.00 .5 0.0 2.0 0.75
i10 5.7 0.2 0.5 7.00 .5 0.8 2.0 0.75
i10 6.0 0.2 0.5 7.00 .5 1.6 2.0 0.75
i10 6.3 0.2 0.5 7.00 .5 2.4 2.0 0.75
i10 6.6 0.2 0.5 7.00 .5 3.2 2.0 0.75
i10 6.9 0.2 0.5 7.00 .5 4.0 2.0 0.75
i10 7.2 0.2 0.5 7.00 .5 0.0 2.0 1.0
i10 7.5 0.2 0.5 7.00 .5 0.8 2.0 1.0
i10 7.8 0.2 0.5 7.00 .5 1.6 2.0 1.0
i10 8.1 0.2 0.5 7.00 .5 2.4 2.0 1.0
i10 8.4 0.2 0.5 7.00 .5 3.2 2.0 1.0
i10 8.7 0.2 0.5 7.00 .5 4.0 2.0 1.0
e

```

```
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Author: Hans Mikelson  
December, 2000 -- January, 2001

New in Csound 4.10

# tempest

tempest — Estimate the tempo of beat patterns in a control signal.

## Description

Estimate the tempo of beat patterns in a control signal.

## Syntax

```
ktemp tempest kin, iprd, imindur, imemdur, ihp, ithresh, ihtim, ixfdbak, \  
istartempo, ifn [, idisprd] [, itweek]
```

## Initialization

*iprd* -- period between analyses (in seconds). Typically about .02 seconds.

*imindur* -- minimum duration (in seconds) to serve as a unit of tempo. Typically about .2 seconds.

*imemdur* -- duration (in seconds) of the *kin* short-term memory buffer which will be scanned for periodic patterns. Typically about 3 seconds.

*ihp* -- half-power point (in Hz) of a low-pass filter used to smooth input *kin* prior to other processing. This will tend to suppress activity that moves much faster. Typically 2 Hz.

*ithresh* -- loudness threshold by which the low-passed *kin* is center-clipped before being placed in the short-term buffer as tempo-relevant data. Typically at the noise floor of the incoming data.

*ihtim* -- half-time (in seconds) of an internal forward-masking filter that masks new *kin* data in the presence of recent, louder data. Typically about .005 seconds.

*ixfdbak* -- proportion of this unit's *anticipated value* to be mixed with the incoming *kin* prior to all processing. Typically about .3.

*istartempo* -- initial tempo (in beats per minute). Typically 60.

*ifn* -- table number of a stored function (drawn left-to-right) by which the short-term memory data is attenuated over time.

*idisprd* (optional) -- if non-zero, display the short-term past and future buffers every *idisprd* seconds (normally a multiple of *iprd*). The default value is 0 (no display).

*itweek* (optional) -- fine-tune adjust this unit so that it is stable when analyzing events controlled by its own output. The default value is 1 (no change).

## Performance

*tempest* examines *kin* for amplitude periodicity, and estimates a current tempo. The input is first low-pass filtered, then center-clipped, and the residue placed in a short-term memory buffer (attenuated over time) where it is analyzed for periodicity using a form of autocorrelation. The period, expressed as a *tempo* in beats per minute, is output as *ktemp*. The period is also used internally to make predictions about future amplitude patterns, and these are placed in a buffer adjacent to that of the input. The two adjacent

buffers can be periodically displayed, and the predicted values optionally mixed with the incoming signal to simulate expectation.

This unit is useful for sensing the metric implications of any k-signal (e.g.- the RMS of an audio signal, or the second derivative of a conducting gesture), before sending to a *tempo* statement.

## Examples

Here is an example of the *tempo* opcode. It uses the file *tempest.csd* [examples/tempest.csd], and *beats.wav* [examples/beats.wav].

### Example 998. Example of the *tempo* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      -iadc      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o tempest.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Use the "beats.wav" sound file.
asig soundin "beats.wav"
; Extract the pitch and the envelope.
kcps, krms pitchamdf asig, 150, 500, 200

iprd = 0.01
imindur = 0.1
imemdur = 3
ihp = 1
ithresh = 30
ihtim = 0.005
ixfdbak = 0.05
istartempo = 110
ifn = 1

; Estimate its tempo.
k1 tempo krms, iprd, imindur, imemdur, ihp, ithresh, ihtim, ixfdbak, istartempo, ifn
printk2 k1

out asig
endin

</CsInstruments>
<CsScore>

; Table #1, a declining line.
f 1 0 128 16 1 128 1
```

```
; Play Instrument #1 for two seconds.  
i 1 0 2  
e
```

```
</CsScore>  
</CsoundSynthesizer>
```

The tempo of the audio file “beats.wav” is 120 beats per minute. In this examples, tempest will print out its best guess as the audio file plays. Its output should include lines like this:

```
. i1 118.24654  
. i1 121.72949
```

# tempo

tempo — Apply tempo control to an uninterpreted score.

## Description

Apply tempo control to an uninterpreted score.

## Syntax

```
tempo ktempo, istartempo
```

## Initialization

*istartempo* -- initial tempo (in beats per minute). Typically 60.

## Performance

*ktempo* -- The tempo to which the score will be adjusted.

*tempo* allows the performance speed of Csound scored events to be controlled from within an orchestra. It operates only in the presence of the Csound *-t* flag. When that flag is set, scored events will be performed from their uninterpreted p2 and p3 (beat) parameters, initially at the given command-line tempo. When a *tempo* statement is activated in any instrument (*ktempo* 0.), the operating tempo will be adjusted to *ktempo* beats per minute. There may be any number of *tempo* statements in an orchestra, but coincident activation is best avoided.

## Examples

Here is an example of the tempo opcode. Remember, it only works if you use the *-t* flag with Csound. The example uses the file *tempo.csd* [examples/tempo.csd].

### Example 999. Example of the tempo opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      -t60 ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o tempo.wav -W -t60 ;;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1
```

```
; Instrument #1.
instr 1
  kval tempoval

  printk 0.1, kval

  ; If the fourth p-field is 1, increase the tempo.
  if (p4 == 1) kgoto speedup
    kgoto playit

speedup:
  ; Increase the tempo to 150 beats per minute.
  tempo 150, 60

playit:

  a1 oscil 10000, 440, 1
  out a1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; p4 = plays at a faster tempo (when p4=1).
; Play Instrument #1 at the normal tempo, repeat 3 times.
r3
i 1 00.00 00.25 0
i 1 00.25 00.25 0
i 1 00.50 00.25 0
i 1 00.75 00.25 0
s

; Play Instrument #1 at a faster tempo, repeat 3 times.
r3
i 1 00.00 00.25 1
i 1 00.25 00.25 0
i 1 00.50 00.25 0
i 1 00.75 00.25 0
s

e

</CsScore>
</CsSoundSynthesizer>
```

## See Also

*tempoval*

## Credits

Example written by Kevin Conder.

# temposcal

temposcal — Phase-locked vocoder processing with onset detection/processing, 'tempo-scaling'.

## Description

*temposcal* implements phase-locked vocoder processing using function tables containing sampled-sound sources, with *GEN01*, and *temposcal* will accept deferred allocation tables.

This opcode allows for time and frequency-independent scaling. Time is advanced internally, but controlled by a tempo scaling parameter; when an onset is detected, timescaling is momentarily stopped to avoid smearing of attacks. The quality of the effect is generally improved with phase locking switched on.

*temposcal* will also scale pitch, independently of frequency, using a transposition factor (k-rate).

## Syntax

```
asig temposcal ktimescal, kamp, kpitch, ktab, klock [,ifftsize, idecim, ithresh]
```

## Initialization

*ifftsize* -- FFT size (power-of-two), defaults to 2048.

*idecim* -- decimation, defaults to 4 (meaning hopsize = fftsize/4)

*idbthresh* -- threshold based on dB power spectrum ratio between two successive windows. A detected ratio above it will cancel timescaling momentarily, to avoid smearing (defaults to 1)

## Performance

*ktimescal* -- timescaling ratio, < 1 stretch, > 1 contract.

*kamp* -- amplitude scaling

*kpitch* -- grain pitch scaling (1=normal pitch, < 1 lower, > 1 higher; negative, backwards)

*klock* -- 0 or 1, to switch phase-locking on/off

*ktab* -- source signal function table. Deferred-allocation tables (see *GEN01*) are accepted, but the opcode expects a mono source. Tables can be switched at k-rate.

## Examples

Here is an example of the *temposcal* opcode. It uses the file *temposcal.csd* [examples/temposcal.csd].

### Example 1000. Example of the temposcal opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o temposcal.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ilock = p4
itab = 1
ipitch = 1
iamp = 0.8
ktime linseg 0.3, p3/2, 0.8, p3/2, 0.3
asig temposcal ktime, iamp, ipitch, itab, ilock
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 0 1 "fox.wav" 0 4 0

i 1 0 3.8 0 ;no locking
i 1 4 3.8 1 ;locking
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
February 2010

New plugin in version 5.13

February 2005.

# tempoval

tempoval — Reads the current value of the tempo.

## Description

Reads the current value of the tempo.

## Syntax

```
kres tempoval
```

## Performance

*kres* -- the value of the tempo. If you use a positive value with the *-t* command-line flag, *tempoval* returns the percentage increase/decrease from the original tempo of 60 beats per minute. If you don't, its value will be 60 (for 60 beats per minute).

## Examples

Here is an example of the *tempoval* opcode. Remember, it only works if you use the *-t* flag with Csound. It uses the file *tempoval.csd* [examples/tempoval.csd].

### Example 1001. Example of the *tempoval* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      -iadc  -t60 ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o tempoval.wav -W -t60 ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Adjust the tempo to 120 beats per minute.
tempo 120, 60

; Get the tempo value.
kval tempoval

printks "kval = %f\\n", 0.1, kval
endin
```

```
</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
e
```

```
</CsScore>
</CsoundSynthesizer>
```

Since 120 beats per minute is a 50% increase over the original 60 beats per minute, its output should include lines like:

```
kval = 0.500000
```

## See Also

*tempo* and *miditempo*

## Credits

Example written by Kevin Conder.

New in version 4.15

December 2002. Thanks to Drake Wilson for pointing out unclear documentation.

# tigoto

tigoto — Transfer control at i-time when a new note is being tied onto a previously held note

## Description

Similar to *igoto* but effective only during an i-time pass at which a new note is being “tied” onto a previously held note. (See *i Statement*) It does not work when a tie has not taken place. Allows an instrument to skip initialization of units according to whether a proposed tie was in fact successful. (See also *tival*).

## Syntax

```
tigoto label
```

where *label* is in the same instrument block and is not an expression.

## Examples

Here is an example of the *tigoto* opcode. It uses the file *tigoto.csd* [examples/tigoto.csd].

### Example 1002. Example of the *tigoto* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o tigoto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

idur = abs(p3) ;make p3 positive even if p3 is negative in score
itiv tival
i1 = -1 ;assume this is tied note, so keep fase of oscili
    tigoto slur ;no reinitialisation on tied notes
i1 = 0 ;first note, so reset phase
aatt line p4, idur, 0 ;primary envelope

slur:
    if itiv==0 kgoto note ;no expression on first and second note
aslur linseg 0, idur*.3, p4, idur*.7, 0 ;envelope for slurred note
aatt = aatt + aslur

note:
asig oscili aatt, p5, 1, i1
outs asig, asig
```

```
endin
</CsInstruments>
<CsScore>
f1 0 4096 10 1 ;sine wave

i1 0 -5 .8 451 ;p3 = 5 seconds
i1 1.5 -1.5 .1 512
i1 3 2 .7 440 ;3 notes together--> duration = 5 seconds

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*cigoto, goto, if, igoto, kgoto, timeout*

More on this opcode: <http://www.csoundjournal.com/2005fall/tiedNotes.html> , written by Steven Yi

# timedseq

timedseq — Time Variant Sequencer

## Description

An event-sequencer in which time can be controlled by a time-pointer. Sequence data are stored into a table.

## Syntax

```
ktrig timedseq ktimepnt, ifn, kp1 [,kp2, kp3, ...,kpN]
```

## Initialization

*ifn* -- number of table containing sequence data.

## Performance

*ktri* -- output trigger signal

*ktimepnt* -- time pointer into sequence file, in seconds.

*kp1,...,kpN* -- output p-fields of notes. *kp2* meaning is relative action time and *kp3* is the duration of notes in seconds.

*timedseq* is a sequencer that allows to schedule notes starting from a user sequence, and depending from an external timing given by a time-pointer value (*ktimepnt* argument). User should fill table *ifn* with a list of notes, that can be provided in an external text file by using GEN23, or by typing it directly in the orchestra (or score) file with GEN02. The format of the text file containing the sequence is made up simply by rows containing several numbers separated by space (similarly to normal Csound score). The first value of each row must be a positive or null value, except for a special case that will be explained below. This first value is normally used to define the instrument number corresponding to that particular note (like normal score). The second value of each row must contain the action time of corresponding note and the third value its duration. This is an example:

```
0 0      0.25 1  93
0 0.25  0.25 2  63
0 0.5   0.25 3  91
0 0.75  0.25 4  70
0 1     0.25 5  83
0 1.25  0.25 6  75
0 1.5   0.25 7  78
0 1.75  0.25 8  78
0 2     0.25 9  83
0 2.25  0.25 10 70
0 2.5   0.25 11 54
0 2.75  0.25 12 80
-1 3    -1   -1 -1 ;; last row of the sequence
```

In this example, the first value of each row is always zero (it is a dummy value, but this p-field can be used, for example, to express a MIDI channel or an instrument number), except the last row, that begins with -1. This value (-1) is a special value, that indicates the end of sequence. It has itself an action time, because sequences can be looped. So the previous sequence has a default duration of 3 seconds, being value 3 the last action time of the sequence.

It is important that ALL lines contains the same number of values (in the example all rows contains exactly 5 values). The number of values contained by each row, MUST be the number of kpXX output arguments (notice that, even if kp1, kp2 etc. are placed at the right of the opcode, they are output arguments, not input arguments).

ktimpnt argument provide the real temporization of the sequence. Actually the passage of time through sequence is specified by ktimpnt itself, which represents the time in seconds. ktimpnt must always be positive, but can move forwards or backwards in time, be stationary or discontinuous, as a pointer into the sequence file, in the same way of pvoc or lpread. When ktimpnt crosses the action time of a note, a trigger signal is sent to ktrig output argument, and kp1, kp2,...kpN arguments are updated with the values of that note. This information can then be used with schedkwhen to actually activate note events. Notice that kp1,...kpN data can be further processed (for example delayed with delayk, transposed, etc.) before feeding schedkwhen.

*ktimpnt* can be controlled by a linear signal, for example:

```
ktimpnt line      0, p3, 3 ; original sequence duration was 3 secs
ktrig  timedseq ktimpnt, 1, kp1, kp2, kp3, kp4, kp5
      schedkwhen ktrig, 105, 2, 0, kp3, kp4, kp5
```

in this case the complete sequence (with original duration of 3 seconds) will be played in *p3* seconds.

You can loop a sequence by controlling it with a phasor:

```
kphs  phasor  1/3
ktimpnt =      kphs * 3
ktrig  timedseq ktimpnt ,1 ,kp1, kp2, kp3, kp4, kp5
      schedkwhen ktrig, 105, 2, 0, kp3, kp4, kp5
```

Obviously you can play only a fragment of the sequence, read it backward, and non-linearly access sequence data in the same way of pvoc and lpread opcodes.

With timedseq opcode you can do almost all things of a normal score, except you have the following limitations:

1. You can't have two notes exactly starting with the same action time; actually at least a k-cycle should separate timing of two notes (otherwise the schedkwhen mechanism eats one of them).
2. All notes of the sequence must have the same number of p-fields (even if they activate different instruments).

You can remedy this limitation by filling with dummy values notes that belongs to instruments with less p-fields than other ones.

## Examples

Here is a complete example of the timedseq opcode. It uses the file *timedseq.csd* [examples/timedseq.csd].

### Example 1003. Example of the timedseq opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
```

```

-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o timedseq.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giseq ftgen 0,0,128,-2, 2, 0, 0.5, 8.00,\ ;first note
          2, 1, 0.5, 8.02,\ ;second note
          2, 2, 0.5, 8.04,\ ;third
          2, 3, 0.5, 8.05,\ ;fourth
          2, 4, 0.5, 8.07,\ ;fifth
          2, 5, 0.5, 8.09,\ ;sixth
          2, 6, 0.5, 8.11,\ ;seventh
          2, 7, 0.5, 9.00,\ ;eight note
          2, 8, 0.5, 8.00,\ ;due to a quirk in the opcode, it needs an extra note - a
          -1, 8, -1, -1 ;last line is a dummy event that indicates to timedseq when t

instr 1

ibeats = 8      ;lengths of sequence in beats
itempo = p4     ;tempo
iBPS = itempo/60 ;beats per second
kphase phasor iBPS/ibeats ;phasor to move through table
kpointer = kphase*ibeats ;multiply phase (range 0 - 1) by the number of beats contained within the s
kp1 init 0
kp2 init 0
kp3 init 0
kp4 init 0
ktrigger timedseq kpointer, giseq, kp1, kp2,kp3, kp4
schedkwhen ktrigger, 0, 0, 2, 0, kp3/abs(iBPS), kp4 ;p3 values have been scaled according to tempo so t
endin ;abs(iBPS)(absolute value) is used because the tempo provided by the fourth note of the sco
; Durations here should be positive, because negative values for duration would indicate a held n
instr 2

aenv linseg 0,0.01,1,p3-0.01,0 ;amplitude envelope
asig vco2 0.4, cpspch(p4), 4, 0.5
outs asig*aenv, asig*aenv
endin

</CsInstruments>
<CsScore>
i 1 0 4 120
i 1 + . 240
i 1 + . 480
i 1 + . -480 ;when negative it plays backwards
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*GEN02, GEN23, seqtime, seqtime2, trigseq*

## Credits

Author: Gabriel Maldonado

# timeinstk

timeinstk — Read absolute time in k-rate cycles.

## Description

Read absolute time, in k-rate cycles, since the start of an instance of an instrument. Called at both i-time as well as k-time.

## Syntax

```
kres timeinstk
```

## Performance

*timeinstk* is for time in k-rate cycles. So with:

```
sr    = 44100
kr    = 6300
ksmps = 7
```

then after half a second, the *timeinstk* opcode would report 3150. It will always report an integer.

*timeinstk* produces a k-rate variable for output. There are no input parameters.

*timeinstk* is similar to *timek* except it returns the time since the start of this instance of the instrument.

## Examples

Here is an example of the *timeinstk* opcode. It uses the file *timeinstk.csd* [examples/timeinstk.csd].

### Example 1004. Example of the *timeinstk* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o timeinstk.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
```

```
    ; Print out the value from timeinstk every half-second.
    k1 timeinstk
    printks "k1 = %f samples\\n", 0.5, k1
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for two seconds.
i 1 0 2
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
k1 = 1.000000 samples
k1 = 2205.000000 samples
k1 = 4410.000000 samples
k1 = 6615.000000 samples
k1 = 8820.000000 samples
```

## See Also

*timeinsts, timek, times*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

# timeinsts

timeinsts — Read absolute time in seconds.

## Description

Read absolute time, in seconds, since the start of an instance of an instrument.

## Syntax

```
kres timeinsts
```

## Performance

Time in seconds is available with *timeinsts*. This would return 0.5 after half a second.

*timeinsts* produces a k-rate variable for output. There are no input parameters.

*timeinsts* is similar to *times* except it returns the time since the start of this instance of the instrument.

## Examples

Here is an example of the *timeinsts* opcode. It uses the file *timeinsts.csd* [examples/timeinsts.csd].

### Example 1005. Example of the *timeinsts* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
-iadc      ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o timeinsts.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine ftgen 0, 0, 2^10, 10, 1

instr 1

kvib init 1
ktim timeinsts ;read time

if ktim > 2 then ;do something after 2 seconds
  kvib oscili 2, 3, giSine ;make a vibrato
endif

asig poscil .5, 600+kvib, giSine ;add vibrato
```

```
    outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 5
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*timeinstk, timek, times*

## Credits

Author: Robin Whittle  
Australia  
May 1997

# timek

timek — Read absolute time in k-rate cycles.

## Description

Read absolute time, in k-rate cycles, since the start of the performance.

## Syntax

```
ires timek
```

```
kres timek
```

## Performance

*timek* is for time in k-rate cycles. So with:

```
sr    = 44100
kr    = 6300
ksmps = 7
```

then after half a second, the *timek* opcode would report 3150. It will always report an integer.

*timek* can produce a k-rate variable for output. There are no input parameters.

*timek* can also operate only at the start of the instance of the instrument. It produces an i-rate variable (starting with *i* or *gi*) as its output.

## Examples

Here is an example of the *timek* opcode. It uses the file *timek.csd* [examples/timek.csd].

### Example 1006. Example of the *timek* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc    ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o timek.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1
```

```
; Instrument #1.
instr 1
; Print out the value from timek every half-second.
k1 timek
printks "k1 = %f samples\\n", 0.5, k1
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for two seconds.
i 1 0 2
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
k1 = 1.000000 samples
k1 = 2205.000000 samples
k1 = 4410.000000 samples
k1 = 6615.000000 samples
k1 = 8820.000000 samples
```

## See Also

*timeinstk, timensts, times*

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.47

Example written by Kevin Conder.

# times

times — Read absolute time in seconds.

## Description

Read absolute time, in seconds, since the start of the performance.

## Syntax

```
ires times
```

```
kres times
```

## Performance

Time in seconds is available with *times*. This would return 0.5 after half a second.

*times* can both produce a k-rate variable for output. There are no input parameters.

*times* can also operate at the start of the instance of the instrument. It produces an i-rate variable (starting with *i* or *gi*) as its output.

## Examples

Here is an example of the times opcode. It uses the file *times\_complex.csd* [examples/times\_complex.csd].

### Example 1007. Example of the times opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o times_complex.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
;by joachim heintz and rory walsh
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giWave   ftgen      0, 0, 1024, 10, 1, .5, .25

instr again

instance =          p4
;reset the duration of this instance
iDur      rnd31     5, 3      ;shorter values are more probable
```

```

iDur      =      abs(iDur) + 0.2
p3        =      iDur
;trigger the effect instrument of this instance
event_i   "i", "fx_processor", 0, iDur, instance
;print the status quo
kTime     times
prints    "instance = %d, start = %f, duration = %f\n", instance, i(kTime), iDur
;make sound
iamp      active 1      ;scale amplitudes
iOct      random 5, 10  ;find pitch
aEnv      transeg 0, 0.02, 0, 1/iamp, p3-0.02, -6, 0 ;output envelope
aSend     poscil aEnv, cpsoct(iOct), giWave ;audio signal
;send signal to effect instrument
Sbus      sprintf "audio_%d", instance ;create unique software bus
          chnset aSend/2, Sbus ;send audio on this bus
;get the last k-cycle of this instance and trigger the successor in it
kLast     release
          schedkwhen kLast, 0, 0, "again", 0, 1, instance+1
endin

instr fx_processor
;apply feedback delay to the above instrument
iwhich    =      p4 ;receive instance number ...
Sbus      sprintf "audio_%d", iwhich ; ... and related software bus
audio     chnget Sbus ;receive audio on this bus
irvbtim   random 1, 5 ;find reverb time
p3        =      p3+irvbtim ;adjust instrument duration
iltptmL   random .1, .5 ;find looptime left ...
iltptmR   random .1, .5 ;...and right
ipan      random 0, 1 ;pan and ...
imix      random 0, 1 ;... mix audio
aL,aR     pan2 audio, ipan ;create stereo
awetL     comb aL, irvbtim, iltptmL ;comb filter
awetR     comb aR, irvbtim, iltptmR
aoutL     ntrpol aL, awetL, imix ;wet-dry mix
aoutR     ntrpol aR, awetR, imix
outs      aoutL/2, aoutR/2
endin

</CsInstruments>
<CsScore>
i "again" 0 1 1

e 3600
</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like these:

```

new alloc for instr again:
instance = 1, start = 0.000000, duration = 0.650439
new alloc for instr fx_processor:
instance = 2, start = 0.650884, duration = 0.411043
new alloc for instr fx_processor:
instance = 3, start = 1.061587, duration = 0.231085
new alloc for instr fx_processor:
instance = 4, start = 1.292336, duration = 0.543473
new alloc for instr fx_processor:
instance = 5, start = 1.835828, duration = 1.777097

```

## See Also

*timeinstk, timeinsts, timek*

## Credits

Author: Robin Whittle  
Australia  
May 1997

# timeout

timeout — Conditional branch during p-time depending on elapsed note time.

## Description

Conditional branch during p-time depending on elapsed note time. *istrt* and *idur* specify time in seconds. The branch to *label* will become effective at time *istrt*, and will remain so for just *idur* seconds. Note that *timeout* can be reinitialized for multiple activation within a single note (see example under *reinit*).

## Syntax

```
timeout istrt, idur, label
```

where *label* is in the same instrument block and is not an expression.

## Examples

Here is an example of the timeout opcode. It uses the file *timeout.csd* [examples/timeout.csd].

### Example 1008. Example of the timeout opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o timeout.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

indx = 0
itim = p4      ;change time for one step

clock:
  timeout 0, itim, time
  reinit clock

time:
  itmp table indx, 2, 0, 0, 1
  if itmp == 1 then
    print itmp
    event_i "i",2, 0, .1      ;event has duration of .1 second
  endif
  indx = indx+1
```

```
    endin

    instr 2 ;play it

    kenv transeg 0.01, p3*0.25, 1, 1, p3*0.75, .5, 0.01
    asig oscili kenv*.4, 400, 1
         outs asig, asig

    endin
</CsInstruments>
<CsScore>
f 1 0 1024 10 1 ;sine
f 2 0 16 2 1 0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 ;the rythm table

i1 0 10 .1
i1 + 10 .05
i1 + 10 .01
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*goto, if, igoto, kgoto, tigo*

# tival

tival — Puts the value of the instrument's internal “tie-in” flag into the named i-rate variable.

## Syntax

```
ir tival
```

## Description

Puts the value of the instrument's internal “tie-in” flag into the named i-rate variable.

## Initialization

Puts the value of the instrument's internal “tie-in” flag into the named i-rate variable. Assigns 1 if this note has been “tied” onto a previously held note (see *i statement*); assigns 0 if no tie actually took place. (See also *tigoto*.)

## Examples

Here is an example of the tival opcode. It uses the file *tival.csd* [examples/tival.csd].

### Example 1009. Example of the tival opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o tival.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

idur = abs(p3)   ;make p3 positive even if p3 is negative in score
itiv tival
i1 = -1   ;assume this is tied note, so keep fase of oscili
        tigoto slur   ;no reinitialisation on tied notes
i1 = 0   ;first note, so reset phase
aatt line p4, idur, 0   ;primary envelope

slur:
if itiv==0 kgoto note   ;no expression on first and second note
aslur linseg 0, idur*.3, p4, idur*.7, 0 ;envelope for slurred note
aatt = aatt + aslur
```

```
note:
asig oscili aatt, p5, 1, i1
     outs  asig, asig

endin
</CsInstruments>
<CsScore>
f1 0 4096 10 1 ;sine wave

i1 0 -5 .8 451 ;p3 = 5 seconds
i1 1.5 -1.5 .1 512
i1 3 2 .7 440 ;3 notes together--> duration = 5 seconds

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*=, divz, init*

More on this opcode: <http://www.csoundjournal.com/2005fall/tiedNotes.html> , written by Steven Yi

# tlineto

tlineto — Generate glissandos starting from a control signal.

## Description

Generate glissandos starting from a control signal with a trigger.

## Syntax

```
kres tlineto ksig, ktime, ktrig
```

## Performance

*kres* -- Output signal.

*ksig* -- Input signal.

*ktime* -- Time length of glissando in seconds.

*ktrig* -- Trigger signal.

*tlineto* is similar to *lineto* but can be applied to any kind of signal (not only stepped signals) without producing discontinuities. Last value of each segment is sampled and held from input signal each time *ktrig* value is set to a nonzero value. Normally *ktrig* signal consists of a sequence of zeroes (see *trigger opcode*).

The effect of glissando is quite different from *port*. Since in these cases, the lines are straight. Also the context of usage is different.

## Examples

Here is an example of the tlineto opcode. It uses the file *tlineto.csd* [examples/tlineto.csd].

### Example 1010. Example of the tlineto opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o tlineto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine ftgen 0, 0, 2^10, 10, 1
```

```
instr 1

kmtr lfo 1, .5, 1 ;produce trigger signal
ktr trigger kmtr, .5, 0 ;with triangle wave

ktime = p4
kfreq randh 1000, 3, .2, 0, 500 ;generate random values
kfreq tlineto kfreq, ktime, ktr ;different glissando times
aout poscil .4, kfreq, giSine
outs aout, aout

endin
</CsInstruments>
<CsScore>

i 1 0 10 .2 ;short glissando
i 1 11 10 .8 ;longer glissande
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*lineto*

## Credits

Author: Gabriel Maldonado

New in Version 4.13

# tone

tone — A first-order recursive low-pass filter with variable frequency response.

## Description

A first-order recursive low-pass filter with variable frequency response.

*tone* is a 1 term IIR filter. Its formula is:

$$y_n = c1 * x_n + c2 * y_{n-1}$$

where

- $b = 2 - \cos(2 \pi \text{ hpsr})$ ;
- $c2 = b - \sqrt{b^2 - 1.0}$
- $c1 = 1 - c2$

## Syntax

```
ares tone asig, khp [, iskip]
```

## Initialization

*iskip* (optional, default=0) -- initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*ares* -- the output audio signal.

*asig* -- the input audio signal.

*khp* -- the response curve's half-power point, in Hertz. Half power is defined as peak power / root 2.

*tone* implements a first-order recursive low-pass filter in which the variable *khp* (in Hz) determines the response curve's half-power point. Half power is defined as peak power / root 2.

## Examples

Here is an example of the tone opcode. It uses the file *tone.csd* [examples/tone.csd].

### Example 1011. Example of the tone opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
```

```
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o tone.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 2

instr 1

asig diskin2 "beats.wav", 1
outs asig, asig
endin

instr 2

kton line 10000, p3, 0 ;all the way down to 0 Hz
asig diskin2 "beats.wav", 1
asig tone asig, kton ;half-power point at 500 Hz
outs asig, asig
endin

</CsInstruments>
<CsScore>

i 1 0 2
i 2 2 2

e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*areson, aresonk, atone, atonek, port, portk, reson, resonk, tonek*

# tonek

tonek — A first-order recursive low-pass filter with variable frequency response.

## Description

A first-order recursive low-pass filter with variable frequency response.

## Syntax

```
kres tonek ksig, khp [, iskip]
```

## Initialization

*iskip* (optional, default=0) -- initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*kres* -- the output signal at control-rate.

*ksig* -- the input signal at control-rate.

*khp* -- the response curve's half-power point, in Hertz. Half power is defined as peak power / root 2.

*tonek* is like *tone* except its output is at control-rate rather than audio rate.

## Examples

Here is an example of the tonek opcode. It uses the file *tonek.csd* [examples/tonek.csd].

### Example 1012. Example of the tonek opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsoundOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o tonek.wav -W ;; for file output any platform
</CsoundOptions>
<CsoundInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gisin ftgen 0, 0, 2^10, 10, 1
```

```
instr 1

ksig randomh 400, 1800, 150
aout poscil .2, 100+ksig, gisin
outs aout, aout
endin

instr 2

ksig randomh 400, 1800, 150
khp line 1, p3, 100 ;vary high-pass
ksig tonek ksig, khp
aout poscil .2, 100+ksig, gisin
outs aout, aout
endin

</CsInstruments>
<CsScore>

i 1 0 5
i 2 5.5 5
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*areson, aresonk, atone, atonek, port, portk, reson, resonk, tone*

## Credits

Author: Robin Whittle  
Australia  
May 1997

# tonex

tonex — Emulates a stack of filters using the tone opcode.

## Description

*tonex* is equivalent to a filter consisting of more layers of *tone* with the same arguments, serially connected. Using a stack of a larger number of filters allows a sharper cutoff. They are faster than using a larger number instances in a Csound orchestra of the old opcodes, because only one initialization and k- cycle are needed at time and the audio loop falls entirely inside the cache memory of processor.

## Syntax

```
ares tonex asig, khp [, inumlayer] [, iskip]
ares tonex asig, ahp [, inumlayer] [, iskip]
```

## Initialization

*inumlayer* (optional) -- number of elements in the filter stack. Default value is 4.

*iskip* (optional, default=0) -- initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*asig* -- input signal

*khp/ahp* -- the response curve's half-power point. Half power is defined as peak power / root 2.

## Examples

Here is an example of the tonex opcode. It uses the file *tonex.csd* [examples/tonex.csd].

### Example 1013. Example of the tonex opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o tonex.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
odbf = 1
```

```
nchnls = 2

instr 1
asig diskin2 "beats.wav", 1
    outs asig, asig
endin

instr 2

kton line 10000, p3, 0 ;all the way down to 0 Hz
asig diskin2 "beats.wav", 1
asig tonex asig, kton, 8 ;8 filters
    outs asig, asig
endin
</CsInstruments>
<CsScore>

i 1 0 2
i 2 3 2

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*atonex, resonx*

## Credits

Author: Gabriel Maldonado (adapted by John ffitch)  
Italy

New in Csound version 3.49

Audio rate parameters introduced in version 6.02

October 2013.

# trandom

trandom — Generates a controlled pseudo-random number series between min and max values according to a trigger.

## Description

Generates a controlled pseudo-random number series between min and max values at k-rate whenever the trigger parameter is different to 0.

## Syntax

```
kout trandom ktrig, kmin, kmax
```

## Performance

*ktrig* -- trigger (opcode produces a new random number whenever this value is not 0).

*kmin* -- minimum range limit

*kmax* -- maximum range limit

*trandom* is almost identical to *random* opcode, except *trandom* updates its output with a new random value only when the *ktrig* argument is triggered (i.e. whenever it is not zero).

## Examples

Here is an example of the trandom opcode. It uses the file *trandom.csd* [examples/trandom.csd].

### Example 1014. Example of the trandom opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o trandom.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

seed 0      ; every run time different values

instr 1

kmin init 0 ;random number between 0 and 220
kmax init 220
```

```
ktrig = p4
k1  trandom ktrig, kmin, kmax
    printk2 k1 ;print when k1 changes
asig poscil .4, 220+k1, 1 ;if triggered, add random values to frequency
    outs asig, asig

endin
</CsInstruments>
<CsScore>
f1 0 4096 10 1

i 1 0 2 0 ;not triggered
i 1 + 2 1 ;triggered
e
</CsScore>
</CsoundSynthesizer>
```

## See also

*random*

## Credits

Written by Gabriel Maldonado.

New in Csound 5.06

# tradsyn

tradsyn — Streaming partial track additive synthesis

## Description

The *tradsyn* opcode takes an input containing a TRACKS pv streaming signal (as generated, for instance by *partials*), as described in Lazzarini et al, "Time-stretching using the Instantaneous Frequency Distribution and Partial Tracking", Proc.of ICMC05, Barcelona. It resynthesises the signal using linear amplitude and frequency interpolation to drive a bank of interpolating oscillators with amplitude and pitch scaling controls.

## Syntax

```
asig tradsyn fin, kscal, kpitch, kmaxtracks, ifn
```

## Performance

*asig* -- output audio rate signal

*fin* -- input pv stream in TRACKS format

*kscal* -- amplitude scaling

*kpitch* -- pitch scaling

*kmaxtracks* -- max number of tracks in resynthesis. Limiting this will cause a non-linear filtering effect, by discarding newer and higher-frequency tracks (tracks are ordered by start time and ascending frequency, respectively)

*ifn* -- function table containing one cycle of a sinusoid (sine or cosine).

## Examples

Here is an example of the *tradsyn* opcode. It uses the file *tradsyn.csd* [examples/tradsyn.csd].

### Example 1015. Example of the *tradsyn* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o tradsyn.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
```

```
nchnls = 2
odbfs = 1

instr 1

ipch = p4
ain diskin2 "beats.wav", 1
fsl,fsi2 pvsifd ain,2048,512,1 ; ifd analysis
fst partials fsl,fsi2,.003,1,3,500 ; partial tracking
aout tradsyn fst, 1, ipch, 500, 1 ; resynthesis
outs aout, aout

endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1

i 1 0 2 1.5 ;up a 5th
i 1 + 2 .5 ;octave down
e
</CsScore>
</CsSoundSynthesizer>
```

The example above shows partial tracking of an ifd-analysis signal and linear additive resynthesis with pitch shifting.

## Credits

Author: Victor Lazzarini  
June 2005

New plugin in version 5

November 2004.

## transeg

transeg — Constructs a user-definable envelope.

### Description

Constructs a user-definable envelope.

### Syntax

```
ares transeg ia, idur, itype, ib [, idur2] [, itype] [, ic] ...
kres transeg ia, idur, itype, ib [, idur2] [, itype] [, ic] ...
```

### Initialization

*ia* -- starting value.

*ib*, *ic*, etc. -- value after *idur* seconds.

*idur* -- duration in seconds of first segment. A zero or negative value will cause all initialization to be skipped.

*idur2*,... *idurx* etc. -- duration in seconds of segment

*itype*, *itype2*, etc. -- if 0, a straight line is produced. If non-zero, then *transeg* creates the following curve, for *n* steps:

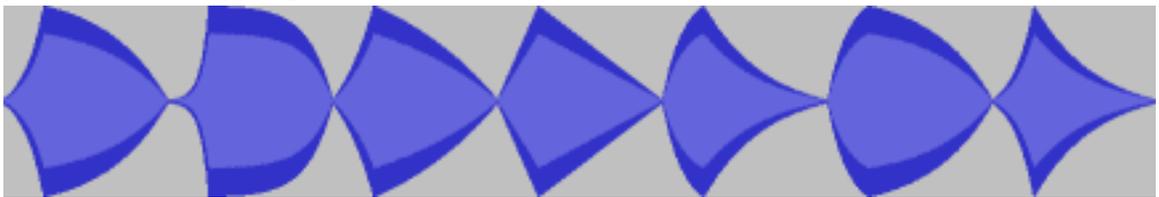
$$ibeg + (ivalue - ibeg) * (1 - \exp(-i*itype/(n-1))) / (1 - \exp(-itype))$$

### Performance

If *itype* > 0, there is a slowly rising (concave) or slowly decaying (convex) curve, while if *itype* < 0, the curve is fast rising (convex) or fast decaying (concave). See also *GEN16*.

### Examples

Here is an example of the transeg opcode. It uses the file *transeg.csd* [examples/transeg.csd]. The example produces the following output:



Output of the transeg example.

## Example 1016. Example of the transeg opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac      -iadc      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o transeg.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 128
nchnls = 2

0dbfs = 1

instr 1
; p4 and p5 determine the type of curve for each
; section of the envelope
kenv transeg 0.01, p3*0.25, p4, 1, p3*0.75, p5, 0.01
a1 oscil kenv, 440, 1
outs a1, a1
endin

</CsInstruments>
<CsScore>
; Table #1, a sine wave.
f 1 0 16384 10 1

i 1 0 2 2 2
i 1 + . 5 5
i 1 + . 1 1
i 1 + . 0 0
i 1 + . -2 -2
i 1 + . -2 2
i 1 + . 2 -2
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*expsega, expsegr, linseg, linsegr, transegr*

## Credits

Author: John ffitch  
University of Bath, Codemist. Ltd.  
Bath, UK  
October 2000

New in Csound version 4.09

Thanks goes to Matt Gerassimoff for pointing out the correct command syntax.

# transegb

transegb — Constructs a user-definable envelope in absolute time.

## Description

Constructs a user-definable envelope in absolute time.

## Syntax

```
ares transegb ia, itim, itype, ib [, itim2] [, itype] [, ic] ...  
kres transegb ia, itim, itype, ib [, itim2] [, itype] [, ic] ...
```

## Initialization

*ia* -- starting value.

*ib*, *ic*, etc. -- value after *itim* seconds.

*itim* -- time in seconds of end of first segment.

*itim2*,... *itimx* etc. -- time in seconds at the end of the segment.

*itype*, *itype2*, etc. -- if 0, a straight line is produced. If non-zero, then *transegb* creates the following curve, for *n* steps:

$$\text{ibeg} + (\text{ivalue} - \text{ibeg}) * (1 - \exp(-i * \text{itype} / (n-1))) / (1 - \exp(\text{itype}))$$

## Performance

If *itype* > 0, there is a slowly rising (concave) or slowly decaying (convex) curve, while if *itype* < 0, the curve is fast rising (convex) or fast decaying (concave). See also *GEN16*.

## Examples

Here is an example of the transegb opcode. It uses the file *transegb.csd* [examples/transegb.csd]. The example produces the following output:

Output of the transegb example.

### Example 1017. Example of the transegb opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform
```

```
; Audio out   Audio in
-odac         -iadc   ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o transeg.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 128
nchnls = 2

0dbfs = 1

instr 1
; p4 and p5 determine the type of curve for each
; section of the envelope
kenv transegb 0.01, p3*0.25, p4, 1, p3, p5, 0.01
a1 oscil kenv, 440, 1
outs a1, a1
endin

</CsInstruments>
<CsScore>
; Table #1, a sine wave.
f 1 0 16384 10 1

i 1 0 2 2 2
i 1 + . 5 5
i 1 + . 1 1
i 1 + . 0 0
i 1 + . -2 -2
i 1 + . -2 2
i 1 + . 2 -2
e
</CsScore>
</CsSoundSynthesizer>
```

## See Also

*expseg, expsega, expsega, expsegr, linseg, linsegb, linsegr, transeg transegr*

## Credits

Author: John ffitch  
University of Bath, Codemist. Ltd.  
Bath, UK  
June 2011

New in Csound version 5.14

# transegr

transegr — Constructs a user-definable envelope with extended release segment.

## Description

Constructs a user-definable envelope. It is the same as *transeg*, with an extended release segment.

## Syntax

```
ares transegr ia, idur, itype, ib [, idur2] [, itype] [, ic] ...  
kres transegr ia, idur, itype, ib [, idur2] [, itype] [, ic] ...
```

## Initialization

*ia* -- starting value.

*ib*, *ic*, etc. -- value after *idur* seconds.

*idur* -- duration in seconds of first segment. A zero or negative value will cause all initialization to be skipped.

*idur2*,... *idurx* etc. -- duration in seconds of segment

*itype*, *itype2*, etc. -- if 0, a straight line is produced. If non-zero, then *transegr* creates the following curve, for *n* steps:

$$\text{ibeg} + (\text{ivalue} - \text{ibeg}) * (1 - \exp(-i * \text{itype} / (n-1))) / (1 - \exp(\text{itype}))$$

## Performance

If *itype* > 0, there is a slowly rising (concave) or slowly decaying (convex) curve, while if *itype* < 0, the curve is fast rising (convex) or fast decaying (concave). See also *GEN16*.

This opcode is the same as of *transeg* with an additional release segment triggered by a MIDI noteoff event, a negative *p1 note event* in the score or a *turnoff2* opcode.

## Examples

Here is an example of the transegr opcode. It uses the file *transegr.csd* [examples/transegr.csd].

### Example 1018. Example of the transegr opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform
```

```
-odac -+rtmidi=virtual -M0 ;;;realtime audio out and realtime midi in
;-iadc ;;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o transegr.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

icps cpsmidi
iamp ampmidi .2
;          st,dur1,typ1,val,dur2,typ2,end
kenv transegr 0, .2, 2, .5, 1, - 3, 0
asig pluck kenv*iamp, icps, icps, 1, 1
      outs asig, asig

endin
</CsInstruments>
<CsScore>
f1 0 4096 10 1 ;sine

f0 30 ;runs 30 seconds
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*expsega, expsegr, linseg, linsegr, transeg*

## Credits

Author: John ffitch  
january 2010

New in Csound version 5.12

# trcross

trcross — Streaming partial track cross-synthesis.

## Description

The *trcross* opcode takes two inputs containing TRACKS pv streaming signals (as generated, for instance by *partials*) and cross-synthesises them into a single TRACKS stream. Two different modes of operation are used: mode 0, cross-synthesis by multiplication of the amplitudes of the two inputs and mode 1, cross-synthesis by the substitution of the amplitudes of input 1 by the input 2. Frequencies and phases of input 1 are preserved in the output. The cross-synthesis is done by matching tracks between the two inputs using a 'search interval'. The matching algorithm will look for tracks in the second input that are within the search interval around each track in the first input. This interval can be changed at the control rate. Wider search intervals will find more matches.

## Syntax

```
fsig trcross fin1, fin2, ksearch, kdepth [, kmode]
```

## Performance

*fsig* -- output pv stream in TRACKS format

*fin1* -- first input pv stream in TRACKS format.

*fin2* -- second input pv stream in TRACKS format

*ksearch* -- search interval ratio, defining a 'search area' around each track of 1st input for matching purposes.

*kdepth* -- depth of effect (0-1).

*kmode* -- mode of cross-synthesis. 0, multiplication of amplitudes (filtering), 1, substitution of amplitudes of input 1 by input 2 (akin to vocoding). Defaults to 0.

## Examples

Here is an example of the *trcross* opcode. It uses the file *trcross.csd* [examples/trcross.csd].

### Example 1019. Example of the *trcross* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o trcross.wav -W ;; for file output any platform
</CsOptions>
```

```
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ain1 diskin2 "beats.wav", 1, 0, 1
ain2 diskin2 "fox.wav", 1

imode = p4
fs1,fsi2 pvsifd ain1, 2048, 512, 1 ; ifd analysis
fst      partials fs1, fsi2, .01, 1, 3, 500 ; partial tracking

fs11,fsi12 pvsifd ain2, 2048, 512, 1 ; ifd analysis (second input)
fst1     partials fs11, fsi12, .01, 1, 3, 500 ; partial tracking (second input)

fcr      trcross fst, fst1, 1.05, 1, imode ; cross-synthesis (mode 0 and mode 1)
aout     tradsyn fcr, 1, 1, 500, 1 ; resynthesis of tracks
outs    aout*3, aout*3

endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1

i 1 0 3 0
i 1 5 3 1

e
</CsScore>
</CsoundSynthesizer>
```

The example above shows partial tracking of two ifd-analysis signals, cross-synthesis, followed by the remix of the two parts of the spectrum and resynthesis.

## Credits

Author: Victor Lazzarini  
February 2006

New in Csound 5.01

# trfilter

trfilter — Streaming partial track filtering.

## Description

The *trfilter* opcode takes an input containing a TRACKS pv streaming signal (as generated, for instance by *partials*) and filters it using an amplitude response curve stored in a function table. The function table can have any size (no restriction to powers-of-two). The table lookup is done by linear-interpolation. It is possible to create time-varying filter curves by updating the amplitude response table with a table-writing opcode.

## Syntax

```
fsig trfilter fin, kamnt, ifn
```

## Performance

*fsig* -- output pv stream in TRACKS format

*fin* -- input pv stream in TRACKS format

*kamnt* -- amount of filtering (0-1)

*ifn* -- function table number. This will contain an amplitude response curve, from 0 Hz to the Nyquist (table indexes 0 to N). Any size is allowed. Larger tables will provide a smoother amplitude response curve. Table reading uses linear interpolation.

## Examples

Here is an example of the trfilter opcode. It uses the file *trfilter.csd* [examples/trfilter.csd].

### Example 1020. Example of the trfilter opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o trfilter.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gifn ftgen 2, 0, -22050, 5, 1, 1000, 1, 4000, 0.000001, 17050, 0.000001 ; low-pass filter curve of 22050
```

```
instr 1

kam line 1, p3, p4
ain disk2 "beats.wav", 1, 0, 1
fs1,fsi2 pvsifd ain, 2048, 512, 1 ; ifd analysis
fst partials fs1, fsi2, .003, 1, 3, 500 ; partial tracking
fsc1 trfilter fst, kam, gifn ; filtering using function table 2
aout tradsyn fsc1, 1, 1, 500, 1 ; resynthesis
outs aout, aout

endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1

i 1 0 4 1
i 1 5 4 0 ;reduce filter effect
e
</CsScore>
</CsoundSynthesizer>
```

The example above shows partial tracking of an ifd-analysis signal and linear additive resynthesis with low-pass filtering.

## Credits

Author: Victor Lazzarini;  
February 2006

New in Csound 5.01

# trhighest

trhighest — Extracts the highest-frequency track from a streaming track input signal.

## Description

The *trhighest* opcode takes an input containing TRACKS pv streaming signals (as generated, for instance by *partials*) and outputs only the highest track. In addition it outputs two k-rate signals, corresponding to the frequency and amplitude of the highest track signal.

## Syntax

```
fsig, kfr, kamp trhighest fin1, kscal
```

## Performance

*fsig* -- output pv stream in TRACKS format

*kfr* -- frequency (in Hz) of the highest-frequency track

*kamp* -- amplitude of the highest-frequency track

*fin* -- input pv stream in TRACKS format.

*kscal* -- amplitude scaling of output.

## Examples

Here is an example of the *trhighest* opcode. It uses the file *trhighest.csd* [examples/trhighest.csd].

### Example 1021. Example of the *trhighest* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o trhighest.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ain      diskin2 "fox.wav", 1
fs1,fsi2 pvsifd ain, 2048, 512, 1 ; ifd analysis
fst partials fs1, fsi2, .1, 1, 3, 500 ; partial tracking
```

```
fhi,kfr,kamp trhighest fst, 1 ; highest freq-track
aout tradsyn fhi, 1, 1, 1, 1 ; resynthesis of highest frequency
outs aout*40, aout*40 ; compensate energy loss

endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1 ;sine wave

i 1 0 3

e
</CsScore>
</CsoundSynthesizer>
```

The example above shows partial tracking of an ifd-analysis signal, extraction of the highest frequency and resynthesis.

## Credits

Author: Victor Lazzarini  
February 2006

New in Csound 5.01

# trigger

trigger — Informs when a krate signal crosses a threshold.

## Description

Informs when a krate signal crosses a threshold.

## Syntax

```
kout trigger ksig, kthreshold, kmode
```

## Performance

*ksig* -- input signal

*kthreshold* -- trigger threshold

*kmode* -- can be 0, 1 or 2

Normally *trigger* outputs zeroes: only each time *ksig* crosses *kthreshold* *trigger* outputs a 1. There are three modes of using *ktrig*:

- *kmode* = 0 - (down-up) *ktrig* outputs a 1 when current value of *ksig* is higher than *kthreshold*, while old value of *ksig* was equal to or lower than *kthreshold*.
- *kmode* = 1 - (up-down) *ktrig* outputs a 1 when current value of *ksig* is lower than *kthreshold* while old value of *ksig* was equal or higher than *kthreshold*.
- *kmode* = 2 - (both) *ktrig* outputs a 1 in both the two previous cases.

## Examples

Here is an example of the trigger opcode. It uses the file *trigger.csd* [examples/trigger.csd].

### Example 1022. Example of the trigger opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o trigger.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
```

```
instr 1

kmtr lfo 1, 1, 1 ;triangle wave
kmode = p4
ktr trigger kmtr, .5, kmode
printk2 ktr
schedkwhen ktr, 0, 3, 2, 0, .3

endin

instr 2

aenv linseg 0,p3*.1,1,p3*.3,1,p3*.6,0 ;envelope
a1 poscil .3*aenv, 1000, 1
outs a1, a1

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine

i 1 0 3 0 ;down-up
i 1 4 3 2 ;down-up & up=down

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.49

# trigseq

`trigseq` — Accepts a trigger signal as input and outputs a group of values.

## Description

Accepts a trigger signal as input and outputs a group of values.

## Syntax

```
trigseq ktrig_in, kstart, kloop, kinitndx, kfn_values, kout1 [, kout2] [...]
```

## Performance

*ktrig\_in* -- input trigger signal

*kstart* -- start index of looped section

*kloop* -- end index of looped section

*kinitndx* -- initial index



### Note

Although *kinitndx* is listed as k-rate, it is in fact accessed only at init-time. So if you are using a k-rate argument, it must be assigned with *init*.

*kfn\_values* -- number of a table containing a sequence of groups of values

*kout1* -- output values

*kout2*, ... (optional) -- more output values

This opcode handles timed-sequences of groups of values stored into a table.

*trigseq* accepts a trigger signal (*ktrig\_in*) as input and outputs group of values (contained in the *kfn\_values* table) each time *ktrig\_in* assumes a non-zero value. Each time a group of values is triggered, table pointer is advanced of a number of positions corresponding to the number of group-elements, in order to point to the next group of values. The number of elements of groups is determined by the number of *koutX* arguments.

It is possible to start the sequence from a value different than the first, by assigning to *kinitndx* an index different than zero (which corresponds to the first value of the table). Normally the sequence is looped, and the start and end of loop can be adjusted by modifying *kstart* and *kloop* arguments. User must be sure that values of these arguments (as well as *kinitndx*) correspond to valid table numbers, otherwise Csound will crash because no range-checking is implemented.

It is possible to disable loop (one-shot mode) by assigning the same value both to *kstart* and *kloop* arguments. In this case, the last read element will be the one corresponding to the value of such arguments. Table can be read backward by assigning a negative *kloop* value.

*trigseq* is designed to be used together with *seqtime* or *trigger* opcodes.

## Examples

Here is an example of the trigseq opcode. It uses the file *trigseq.csd* [examples/trigseq.csd].

### Example 1023. Example of the trigseq opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o trigseq.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giTimes ftgen 91, 0, 128, -2,      1, 1/2, 1/2, 1/8, 1/8, 1/2,1/2, 1/16, 1/16, 1/16, 1/16, 1/16, 1/16, 1
giSeq ftgen 90, 0, 128, -2,      1, 2,      .5, 3,      .25, 4,      .10, 5,      .05, 6 ;** sequence amp

instr 1

icps init p4
iamp init .3

kloop init p5
initndx init p6
kloop2 init p7
initndx2 init p8
kdur init p9
iminTime init p10
imaxTime init p11
kampratio init 1
kfregratio init 1

ktime_unit expseg iminTime,p3/8,iminTime,p3* 3/4,imaxTime,p3/8,imaxTime

;**ktrig seqtime ktime_unit, kstart, kloop, initndx, kfn_times
;ktrig seqtime 1/ktime_unit, 0,      15, 0,      giTimes

ktrig metro ktime_unit

;**** trigseq ktrig_in, kstart, kloop, initndx, kfn_values, kout1 [, kout2, kout3, ..., koutN]
trigseq ktrig, 0, kloop2,initndx2, giSeq,      kampratio, kfregratio

;atrig = ktrig*10000
schedkwhen ktrig, -1, -1, 3, 0, kdur, kampratio*iamp, kfregratio*icps
; schedkwhen ktrig, -1, -1, 2, 0, ktrig, kampratio*iamp, kfregratio*icps
endin

instr 2

icps init p4
iamp init .2

```

```

kloop init p5
initndx init p6
kloop2 init p7
initndx2 init p8
kdur init p9
iminTime init p10
imaxTime init p11
kampratio init 1
kfregratio init 1

ktime_unit expseg iminTime,p3/8,iminTime,p3* 3/4,imaxTime,p3/8,imaxTime

; **ktrig seqtime ktime_unit, kstart, kloop, initndx, kfn_times
ktrig seqtime 1/ktime_unit, 0, 15, 0, giTimes

; ktrig metro ktime_unit

; **** trigseq ktrig_in, kstart, kloop, initndx, kfn_values, kout1 [, kout2, kout3, ..., koutN]
trigseq ktrig, 0, kloop2, initndx2, giSeq, kampratio, kfregratio
printk2 ktrig
; atrig = ktrig*10000
; schedkwhen ktrig, -1, -1, 2, 0, kdur, kampratio*iamp, kfregratio*icps
schedkwhen ktrig, -1, -1, 3, 0, ktrig, kampratio*iamp, kfregratio*icps
endin

instr 3

print p3
kenv expseg 1.04, p3,.04
a1 foscili p4*a(kenv-0.04), p5,1,1,kenv*5, 2
outs a1, a1
endin

</CsInstruments>
<CsScore>
f2 0 8192 10 1

; icps unused unused kloop2 initndx2 kdur iminTime imaxTime

s

i1 0 6 100 0 0 5 0 .2 3 15
i1 8 6 150 0 0 4 1 .1 4 30
i1 16 6 200 0 0 5 3 .25 8 50
i1 24 6 300 0 0 3 0 .1 1 30

i2 32 6 100 0 0 5 0 .2 1 1
i2 40 6 150 0 0 4 1 .1 .5 .5
i2 48 6 200 0 0 5 3 .25 3 .5
i2 56 6 300 0 0 5 0 .1 1 8

e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*seqtime*, *trigger*

## Credits

Author: Gabriel Maldonado

November 2002. Added a note about the *kinindx* parameter, thanks to Rasmus Ekman.

January 2003. Thanks to a note from Øyvind Brandtsegg, I corrected the credits.

New in version 4.06

# trirand

trirand — Triangular distribution random number generator

## Description

Triangular distribution random number generator. This is an x-class noise generator.

## Syntax

```
ares trirand krange
ires trirand krange
kres trirand krange
```

## Performance

*krange* -- the range of the random numbers (*-krange* to *+krange*).

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the trirand opcode. It uses the file *trirand.csd* [examples/trirand.csd].

### Example 1024. Example of the trirand opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o trirand.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; every run time same values

ktri trirand 100
printk .2, ktri ; look
```

```

aout oscili 0.8, 440+ktri, 1 ; & listen
outs aout, aout
endin

instr 2 ; every run time different values

seed 0
ktri trirand 100
printk .2, ktri ; look
aout oscili 0.8, 440+ktri, 1 ; & listen
outs aout, aout
endin

</CsInstruments>
<CsScore>
; sine wave
f 1 0 16384 10 1

i 1 0 2
i 2 3 2
e
</CsScore>
</CsoundSynthesizer>

```

Its output should include a line like this:

```

i 1 time 0.00067: -4.97993
i 1 time 0.20067: 1.20909
i 1 time 0.40067: 17.45873
i 1 time 0.60067: 52.55409
i 1 time 0.80067: -1.92888
i 1 time 1.00000: -11.01149
i 1 time 1.20067: 9.79521
i 1 time 1.40067: 26.98504
i 1 time 1.60067: 24.67405
i 1 time 1.80000: -67.59846
i 1 time 2.00000: 64.24861
WARNING: Seeding from current time 521999639
i 2 time 3.00067: 3.28969
i 2 time 3.20067: 54.98986
i 2 time 3.40067: -33.84788
i 2 time 3.60000: -41.93523
i 2 time 3.80067: -6.61742
i 2 time 4.00000: 39.67097
i 2 time 4.20000: 2.95123
i 2 time 4.40067: 45.59255
i 2 time 4.60067: 16.57259
i 2 time 4.80067: -18.80273
i 2 time 5.00000: -2.01697

```

## See Also

*betarand*, *bexprnd*, *cauchy*, *exprand*, *gauss*, *linrand*, *pcauchy*, *poisson*, *unirand*, *weibull*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

# trlowest

trlowest — Extracts the lowest-frequency track from a streaming track input signal.

## Description

The *trlowest* opcode takes an input containing TRACKS pv streaming signals (as generated, for instance by *partials*) and outputs only the lowest track. In addition it outputs two k-rate signals, corresponding to the frequency and amplitude of the lowest track signal.

## Syntax

```
fsig, kfr, kamp trlowest fin1, kscal
```

## Performance

*f<sub>sig</sub>* -- output pv stream in TRACKS format

*k<sub>fr</sub>* -- frequency (in Hz) of the lowest-frequency track

*k<sub>amp</sub>* -- amplitude of the lowest-frequency track

*f<sub>in</sub>* -- input pv stream in TRACKS format.

*k<sub>scal</sub>* -- amplitude scaling of output.

## Examples

Here is an example of the *trlowest* opcode. It uses the file *trlowest.csd* [examples/trlowest.csd].

### Example 1025. Example of the *trlowest* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o trlowest.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ain      diskin2 "beats.wav", 1
fs1,fsi2 pvsifd ain, 2048, 512, 1 ; ifd analysis
fst partials fs1, fsi2, .003, 1, 3, 500 ; partial tracking
```

```
flow,kfr,kamp trlowest fst, 1 ; lowest freq-track
aout tradsyn flow, 1, 1, 1, 1 ; resynthesis of lowest frequency
outs aout*2, aout*2

endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1 ;sine wave

i 1 0 3
e
</CsScore>
</CsoundSynthesizer>
```

The example above shows partial tracking of an ifd-analysis signal, extraction of the lowest frequency and resynthesis.

## Credits

Author: Victor Lazzarini  
February 2006

New in Csound 5.01

# trmix

trmix — Streaming partial track mixing.

## Description

The *trmix* opcode takes two inputs containing TRACKS pv streaming signals (as generated, for instance by *partials*) and mixes them into a single TRACKS stream. Tracks will be mixed up to the available space (defined by the original number of FFT bins in the analysed signals). If the sum of the input tracks exceeds this space, the higher-ordered tracks in the second input will be pruned.

## Syntax

```
fsig trmix fin1, fin2
```

## Performance

*fsig* -- output pv stream in TRACKS format

*fin1* -- first input pv stream in TRACKS format.

*fin2* -- second input pv stream in TRACKS format

## Examples

Here is an example of the *trmix* opcode. It uses the file *trmix.csd* [examples/trmix.csd].

### Example 1026. Example of the *trmix* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o trmix.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ain diskin2 "fox.wav", 1
fs1,fsi2 pvsifd ain, 2048, 512, 1 ; ifd analysis
fst partials fs1, fsi2, .003, 1, 3, 500 ; partial tracking
fslo,fshi trsplit fst, 1000 ; split partial tracks at 1000 Hz
fsc1 trscale fshi, 1.3 ; shift the upper tracks
fmix trmix fslo,fsc1 ; mix the shifted and unshifted tracks
```

```
aout tradsyn fmix, 1, 1, 500, 1 ; resynthesis of tracks
outs aout, aout

endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1 ;sine wave

i 1 0 3
e
</CsScore>
</CsoundSynthesizer>
```

The example above shows partial tracking of an ifd-analysis signal, frequency splitting and pitch shifting of the upper part of the spectrum, followed by the remix of the two parts of the spectrum and resynthesis.

## Credits

Author: Victor Lazzarini  
February 2006

New in Csound 5.01

# trscale

trscale — Streaming partial track frequency scaling.

## Description

The *trscale* opcode takes an input containing a TRACKS pv streaming signal (as generated, for instance by *partials*) and scales all frequencies by a k-rate amount. It can also, optionally, scale the gain of the signal by a k-rate amount (default 1). The result is pitch shifting of the input tracks.

## Syntax

```
fsig trscale fin, kpitch[, kgain]
```

## Performance

*fsig* -- output pv stream in TRACKS format

*fin* -- input pv stream in TRACKS format

*kpitch* -- frequency scaling

*kgain* -- amplitude scaling (default 1)

## Examples

Here is an example of the *trscale* opcode. It uses the file *trscale.csd* [examples/trscale.csd].

### Example 1027. Example of the *trscale* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o trscale.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kpitch = p4
ain diskin2 "fox.wav", 1
fsl,fsi2 pvsifd ain, 2048, 512, 1 ; ifd analysis
fst partials fsl, fsi2, .003, 1, 3, 500 ; partial tracking
fsc1 trscale fst, kpitch ; frequency scale
```

```

aout tradsyn fscl, 1, 1, 500, 1 ; resynthesis
      outs   aout, aout

endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1

i 1 0 3 1.5 ;up a 5th
i 1 3 3 3 ;two octaves higher
e
</CsScore>
</CsoundSynthesizer>
```

The example above shows partial tracking of an ifd-analysis signal and linear additive resynthesis with pitch shifting.

## Credits

Author: Victor Lazzarini  
February 2006

New in Csound 5.01

# trshift

trshift — Streaming partial track frequency scaling.

## Description

The *trshift* opcode takes an input containing a TRACKS pv streaming signal (as generated, for instance by *partials*) and shifts all frequencies by a k-rate frequency. It can also, optionally, scale the gain of the signal by a k-rate amount (default 1). The result is frequency shifting of the input tracks.

## Syntax

```
fsig trshift fin, kpsft[, kgain]
```

## Performance

*fsig* -- output pv stream in TRACKS format

*fin* -- input pv stream in TRACKS format

*kpsft* -- frequency shift in Hz

*kgain* -- amplitude scaling (default 1)

## Examples

Here is an example of the *trshift* opcode. It uses the file *trshift.csd* [examples/trshift.csd].

### Example 1028. Example of the trshift opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o trshift.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kpsft = p4
ain diskin2 "fox.wav", 1
fs1, fs2 pvsifd ain, 2048, 512, 1 ; ifd analysis
fst partials fs1, fs2, 0.003, 1, 3, 500 ; partial tracking
fsc1 trshift fst, kpsft ; frequency shift
```

```
aout   tradsyn   fscl, 1, 1, 500, 1           ; resynthesis
        outs aout, aout

endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1 ;sine

i 1 0 3 150 ;adds 150Hz to all tracks
i 1 + 3 500 ;adds 500Hz to all tracks
e
</CsScore>
</CsoundSynthesizer>
```

The example above shows partial tracking of an ifd-analysis signal and linear additive resynthesis with frequency shifting.

## Credits

Author: Victor Lazzarini  
February 2006

New in Csound 5.01

# trsplit

trsplit — Streaming partial track frequency splitting.

## Description

The *trsplit* opcode takes an input containing a TRACKS pv streaming signal (as generated, for instance by *partials*) and splits it into two signals according to a k-rate frequency 'split point'. The first output will contain all tracks up from 0Hz to the split frequency and the second will contain the tracks from the split frequency up to the Nyquist. It can also, optionally, scale the gain of the output signals by a k-rate amount (default 1). The result is two output signals containing only part of the original spectrum.

## Syntax

```
fsiglow, fsighi trsplit fin, ksplit[, kgainlow, kgainhigh]
```

## Performance

*fsiglow* -- output pv stream in TRACKS format containing the tracks below the split point.

*fsighi* -- output pv stream in TRACKS format containing the tracks above and including the split point.

*fin* -- input pv stream in TRACKS format

*ksplit* -- frequency split point in Hz

*kgainlow*, *kgainhig* -- amplitude scaling of each one of the outputs (default 1).

## Examples

Here is an example of the *trsplit* opcode. It uses the file *trsplit.csd* [examples/trsplit.csd].

### Example 1029. Example of the *trsplit* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o trsplit.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
```

```
ain disk2 "beats.wav", 1
fs1,fsi2 pvsifd ain, 2048, 512, 1 ; ifd analysis
fst partials fs1, fsi2, .003, 1, 3, 500 ; partial tracking
fslo,fsi trsplit fst, 1500 ; split partial tracks at 1500 Hz
aout tradsyn fshi, 1, 1, 500, 1 ; resynthesis of tracks above 1500Hz
      outs aout, aout

endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1 ;sine

i 1 0 2
e
</CsScore>
</CsoundSynthesizer>
```

The example above shows partial tracking of an ifd-analysis signal and linear additive resynthesis of the upper part of the spectrum (from 1500Hz).

## Credits

Author: Victor Lazzarini  
February 2006

New in Csound 5.01

# turnoff

turnoff — Enables an instrument to turn itself off or to turn an instance of another instrument off.

## Description

Enables an instrument to turn itself off or to turn an instance of another instrument off.

## Syntax

```
turnoff
turnoff inst
turnoff knst
```

## Initialization

## Initialization

*inst* -- when used with an i-time parameter, this is the instance handle of an instrument to be turned off (obtained from the nstance opcode).

*turnoff* -- with no parameters this p-time statement enables an instrument to turn itself off. Whether of finite duration or “held”, the note currently being performed by this instrument is immediately removed from the active note list. No other notes are affected.

*knst* -- when used with an k-time parameter, this is the instance handle of an instrument to be turned off (obtained from the nstance opcode).

## Examples

The following example uses the turnoff opcode. It will cause a note to terminate when a control signal passes a certain threshold (here the Nyquist frequency). It uses the file *turnoff.csd* [examples/turnoff.csd].

### Example 1030. Example of the turnoff opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o turnoff.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
```

```
nchnls = 1

; Instrument #1.
instr 1
  k1 expon 440, p3/10,880 ; begin gliss and continue
  if k1 < sr/2 kgoto contin ; until Nyquist detected
  turnoff ; then quit

contin:
  a1 oscil 10000, k1, 1
  out a1
endin

</CsInstruments>
<CsScore>

; Table #1: an ordinary sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for 4 seconds.
i 1 0 4
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*ihold, turnoff2, turnon*

# turnoff2

turnoff2 — Turn off instance(s) of other instruments at performance time.

## Description

Turn off instance(s) of other instruments at performance time.

## Syntax

```
turnoff2 kinsno, kmode, krelease
```

## Performance

*kinsno* -- instrument to be turned off (can be fractional) if zero or negative, no instrument is turned off

*kmode* -- sum of the following values:

- 0, 1, or 2: turn off all instances (0), oldest only (1), or newest only (2)
- 4: only turn off notes with exactly matching (fractional) instrument number, rather than ignoring fractional part
- 8: only turn off notes with indefinite duration ( $p3 < 0$  or MIDI)

*krelease* -- if non-zero, the turned off instances are allowed to release, otherwise are deactivated immediately (possibly resulting in clicks)



### Note

As a rule of thumb, you should turn off instruments with a higher instrument number than the one where turnoff is called, as doing otherwise might cause initialization issues.

## Examples

The following example uses the turnoff2 opcode, s written by Lou Cohen..

### Example 1031. Example of the turnoff2 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o sin.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```

sr      =      48000      ;samples per second for sound
kr      =      4800
nchnls  =      1
Odbfs   =      32767

gisine    ftgen    1, 0, 131073, 9, 1, 1, 0      ;single sine tone

;-----
instr 2 ;start
  ktrigger    init      0
  if (ktrigger = 0) then
    kMultiple  =      1.1
    kHz        =      440
    kAmp       =      (Odbfs/10)
;startup four instances of instrument 200
    event     "i", 200, 0, 3000, kAmp, kHz
    kAmp      =      kAmp * 0.75
    kHz       =      kHz * kMultiple

    event     "i", 200, 0, 3000, kAmp, kHz
    kAmp      =      kAmp * 0.75
    kHz       =      kHz * kMultiple

    event     "i", 200, 0, 3000, kAmp, kHz
    kAmp      =      kAmp * 0.75
    kHz       =      kHz * kMultiple

    event     "i", 200, 0, 3000, kAmp, kHz
    kAmp      =      kAmp * 0.75
    kHz       =      kHz * kMultiple
    ktrigger   =      1
  endif
endin
;-----
instr 3 ;after 10 seconds, turn off the instruments
  ktrigger    init    1
  if (ktrigger==1) then
    turnoff2 200, 1, 1      ;turn off must recently started instrument instance
    kactive   active 200    ;find out how many are still active
    printk2  kactive      ;print mainly to show progress

    turnoff2 200, 0, 1      ;turn off all the rest of the instruments
    kactive   active 200    ;find out how many are still active
    printk2  kactive, 10    ;print to show progress
  endif
endin
;-----
instr 200 ;play the tone
kEnv    linen    1, 0.1, p3, 0.1
ar      oscil    kEnv*p4, p5, 1
        out      ar
        print    p4, p5

endin
</CsInstruments>
<CsScore>

i2 0 0.1
i3 10 0.1
</CsScore>

</CsoundSynthesizer>

```

## See Also

*turnoff*

## Credits

Author: Istvan Varga  
2005

New in Csound 5.00

# turnon

turnon — Activate an instrument for an indefinite time.

## Description

Activate an instrument for an indefinite time.

## Syntax

```
turnon insnum [, itime]
```

## Initialization

*insnum* -- instrument number to be activated

*itime* (optional, default=0) -- delay, in seconds, after which instrument *insnum* will be activated. Default is 0.

## Performance

*turnon* activates instrument *insnum* after a delay of *itime* seconds, or immediately if *itime* is not specified. Instrument is active until explicitly turned off. (See *turnoff*.)

## See Also

*turnoff*, *turnoff2*

# tvconv

tvconv — A time-varying convolution (FIR filter) opcode.

## Description

An opcode that takes two incoming signals and interprets one of them as the coefficients of linear time-variable finite impulse response filter. This is implemented via direct convolution (for partition sizes of 1 sample) or DFT-based partitioned convolution. The signals can be 'frozen' (i.e. the filter coefficients are kept the same) at any point in time, at a-rate or k-rate.

## Syntax

```
ares tvconv asig1, asig2, xfreez1,  
          xfreez2, iparts, ifils
```

## Initialization

*iparts* -- partition size, for sizes > 1, a DFT-based partitioned convolution process is used. Otherwise a time-domain delay line FIR is implemented. Partition sizes > 1 are rounded to the nearest power-of-two.

*ifils* -- filter size. For partition sizes > 1, filter sizes are rounded to the nearest power-of-two. With partition size = 1, since direct convolution is used, filters can be of any size.

## Performance

*ares* -- audio output.

*asig1, asig2* -- audio inputs.

*xfreez1* -- freeze switch for asig1. Coefficients are only updated (ie. the signal is passing into the convolution) if xfreez1 > 0. This input can take an audio or a k-rate signal, or a constant.

*xfreez2* -- freeze switch for asig2, similar to xfreez1 in operation.

## Examples

Here is an example of the tvconv opcode. It uses the file *tvconv.csd* [examples/tvconv.csd].

### Example 1032. Example of the tvconv opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsoundOptions>  
-odac  
</CsoundOptions>  
<CsoundInstruments>  
  
instr 1  
  asig diskin "fox.wav", 1, 0, 1
```

```
air diskin "beats.wav",1,0,1
air buthp air/0dbfs,1000
k1 linseg 0,p3/3,0,0,1,2*p3/3,1
a1 oscili k1, 0.5, 1
a2 oscili k1, 0.6, 1
asig tvconv asig,air,1-a2,1-a1,256,1024
asig clip asig,1,0dbfs
  out asig
endin

</CsInstruments>
<CsScore>
f1 0 1024 7 0 512 0 1 1 511 1
i1 0 30
</CsScore>
</CsoundSynthesizer>
```

## See also

*pconvolve*, *convolve*, *ftconv*

## Credits

Author: Victor Lazzarini  
2017

New in version 6.09

# unirand

unirand — Uniform distribution random number generator (positive values only).

## Description

Uniform distribution random number generator (positive values only). This is an x-class noise generator.

## Syntax

```
ares unirand krange
```

```
ires unirand krange
```

```
kres unirand krange
```

## Performance

*krange* -- the range of the random numbers (0 - *krange*).

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the unirand opcode. It uses the file *unirand.csd* [examples/unirand.csd].

### Example 1033. Example of the unirand opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o unirand.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; every run time same values

ktri unirand 100
  printk .2, ktri ; look
aout oscili 0.8, 440+ktri, 1 ; & listen
```

```
outs aout, aout
endin

instr 2 ; every run time different values

seed 0
ktri unrand 100
printk .2, ktri ; look
aout oscili 0.8, 440+ktri, 1 ; & listen
outs aout, aout
endin

</CsInstruments>
<CsScore>
; sine wave
f 1 0 16384 10 1

i 1 0 2
i 2 3 2
e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like these:

```
i 1 time 0.00067: 81.47237
i 1 time 0.20067: 41.72671
i 1 time 0.40067: 5.96189
i 1 time 0.60067: 91.59912
i 1 time 0.80067: 85.07127
i 1 time 1.00000: 92.50948
i 1 time 1.20067: 98.79347
i 1 time 1.40067: 98.91449
i 1 time 1.60067: 50.37808
i 1 time 1.80000: 72.02497
i 1 time 2.00000: 52.94362

WARNING: Seeding from current time 4007444022

i 2 time 3.00067: 91.86294
i 2 time 3.20067: 94.68759
i 2 time 3.40067: 1.05825
i 2 time 3.60000: 78.57628
i 2 time 3.80067: 27.67408
i 2 time 4.00000: 76.46347
i 2 time 4.20000: 77.10071
i 2 time 4.40067: 34.28921
i 2 time 4.60067: 37.72286
i 2 time 4.80067: 54.96646
i 2 time 5.00000: 11.67566
B 3.000 .. 5.000 T 5.000 TT 5.000 M: 0.80000 0.80000
Score finished in csoundPerform().
```

## See Also

*seed, betarand, bexprnd, cauchy, exprand, gauss, linrand, pcauchy, poisson, trirand, weibull*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

# until

until — A syntactic looping construction.

## Description

A syntactic looping construction.

## Syntax

```
until condition do
... od
```

## Performance

The statements between the *do* and *od* form the body of a loop which is obeyed until the conditional becomes true.

## Examples

Here is an example of the until construction. It uses the file *until.csd* [examples/until.csd].

### Example 1034. Example of the until opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o ifthen.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 1
lab99:
if p4<0 goto lab100
  p4 = p4-1
  print p4
  goto lab99
lab100:
endin

instr 2
  until p4<0 do
    p4 = p4-1
    print p4
  od
```

```
endin
</CsInstruments>
<CsScore>
i 1 1 1 4
i 2 2 1 4
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
B 0.000 .. 1.000 T 1.000 TT 1.000 M: 0.0
new alloc for instr 1:
instr 1: p4 = 3.000
instr 1: p4 = 2.000
instr 1: p4 = 1.000
instr 1: p4 = 0.000
instr 1: p4 = -1.000
B 1.000 .. 2.000 T 2.000 TT 2.000 M: 0.0
new alloc for instr 2:
instr 2: p4 = 3.000
instr 2: p4 = 2.000
instr 2: p4 = 1.000
instr 2: p4 = 0.000
instr 2: p4 = -1.000
B 2.000 .. 3.000 T 3.000 TT 3.000 M: 0.0
```

## See Also

*loop\_ge*, *loop\_gt* and *loop\_le*. *loop\_lt*. *while*.

## Credits

John ffitch.

New in Csound version 5.14 with new parser

# unwrap

unwrap — Applies a unwrapping operation to an array of phase values.

## Description

Applies a unwrapping operation to a vector of phases stored in an array. The output is an array with phases in the range of  $[-\pi, \pi)$ .

## Syntax

```
kout[] unwrap kin[]
```

## Performance

*kout[]* -- output array containing the unwrapped phases. It will be created if it does not exist.

*kin[]* -- input array containing the input vector.

## Examples

Here is an example of the unwrap opcode. It uses the file *unwrap.csd* [examples/unwrap.csd].

### Example 1035. Example of the unwrap opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
ksmps = 64

ifn1 ftgen 1, 0, 512, 7, 0, 512, 0
ifn2 ftgen 2, 0, 512, 7, 0, 512, 0

opcode PVA,k[]k[]k,aii
  asig, isize, ihop xin
  iolaps init isize/ihop
  kcmt init 0
  krow init 1
  kIn[] init isize
  kOlph[] init isize/2 + 1
  ifac = (sr/(ihop*2*$M_PI));
  iscal = (2*$M_PI*ihop/isize);
  kfl = 0
  kIn shiftn asig
  if kcmt == ihop then
    kWin[] window kIn, krow*ihop
    kSpec[] rfft kWin
    kMags[] mags kSpec
    kPha[] phs kSpec
    kDelta[] = kPha - kOlph
    kOlph = kPha
  kk = 0
```

```

kDelta unwrap kDelta
while kk < isize/2 do
  kPha[kk] = (kDelta[kk] + kk*iscal)*ifac
  kk += 1
od
krow = (krow+1)%iolaps
kcnt = 0
kfl = 1
endif
xout kMags,kPha,kfl
kcnt += ksmps
endop

opcode PVS,a,k[]k[]kii
kMags[],kFr[],kfl,isize,ihop xin
iolaps init isize/ihop
ifac = ihop*2*$M_PI/sr;
iscal = sr/isize
krow init 0
kOla[] init isize
kOut[][] init iolaps,isize
kPhs[] init isize/2+1
if kfl == 1 then
  kk = 0
  while kk < isize/2 do
    kFr[kk] = (kFr[kk] - kk*iscal)*ifac
    kk += 1
  od
  kPhs = kFr + kPhs
  kSpec[] pol2rect kMags,kPhs
  kRow[] rfft kSpec
  kWin[] window kRow, krow*ihop
  kOut setrow kWin, krow
  kOla = 0
  kk = 0
  until kk == iolaps do
    kRow getrow kOut, kk
    kOla = kOla + kRow
    kk += 1
  od
  krow = (krow+1)%iolaps
endif
xout shiftout(kOla)/iolaps
endop

instr 1
isi = 1024
ihop = 128

a1 diskin2 "fox.wav",1,0,1 ; audio input
kMags[],kPhs[],kflg PVA a1,isi,ihop
a2 PVS kMags,kPhs,kflg,isi,ihop
  out a2

endin

</CsInstruments>
<CsScore>
i1 0 10
</CsScore>
</CsoundSynthesizer>

```

## See Also

Vectorial opcodes, array opcodes

## Credits

Author: Victor Lazzarini  
NUI Maynooth  
2014

New in version 6.04

# upsamp

upsamp — Modify a signal by up-sampling.

## Description

Modify a signal by up-sampling.

## Syntax

```
ares upsamp ksig
```

## Performance

*upsamp* converts a control signal to an audio signal. It does it by simple repetition of the kval. *upsamp* is a slightly more efficient form of the assignment, *asig = ksig*.

## Examples

Here is an example of the upsamp opcode. It uses the file *upsamp.csd* [examples/upsamp.csd].

### Example 1036. Example of the upsamp opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o upsamp.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
;;with code from Steven Cook / David Akbari, Menno Knevel and Joachim Heintz

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

seed      0

opcode Decimator, a, akk ;UDO Sample rate / Bit depth reducer
;see http://www.csounds.com/udo/displayOpcode.php?opcode\_id=73
    setksmps 1
ain, kbit, ksrate xin

kbits =      2^kbit          ;bit depth (1 to 16)
kfold =      (sr/ksrate)    ;sample rate
kin =      downsamp ain     ;convert to kr
kin =      (kin+0dbfs)      ;add DC to avoid (-)
kin =      kin*(kbits/(0dbfs*2)) ;scale signal level
kin =      int(kin)         ;quantise
aout =      upsamp kin       ;convert to sr
aout =      aout*(2/kbits)-0dbfs ;rescale and remove DC
a0ut =      fold aout, kfold ;resample
xout =      a0ut
```

```
endop

instr 1 ;avoid playing this too loud

kbit    =      p4
ksr      =      44100
asig     =      diskin "fox.wav", 1
aout     =      Decimator asig, kbit, ksr
          printks "bitrate = %d, ", 3, kbit
          printks "with samplerate = %d\\n", 3, ksr
          outs    aout*.7, aout*.7

endin

instr 2 ;moving randomly between different bit values (1 - 6)

kbit     =      randomi 1, 6, .5, 1
asig     =      diskin "fox.wav", 1, 0, 1 ;loop play
aout     =      Decimator asig, kbit, 44100
          printks "bitrate = %f\\n", .3, kbit
          outs    aout*.7, aout*.7

endin

</CsInstruments>
<CsScore>
i 1 0 3 16 ;sounds allright but
i 1 + 3 5 ;it's getting worse
i 1 + 3 2 ;and worse...
i 2 9 22 ;or quality moves randomly
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*diff, downsamp, integ, interp, samphold*

# urandom

urandom — truly random opcodes with controllable range.

## Description

Truly random opcodes with controllable range. These units are for Unix-like systems only and use `/dev/urandom` to construct Csound random values

## Syntax

```
ax urandom [imin, imax]
ix urandom [imin, imax]
kx urandom [imin, imax]
```

## Initialization

*ix* -- i-rate output value.

*imin* -- minimum value of range; defaults to -1.

*imax* -- maximum value of range; defaults to +1.



### Notes

The algorithm produces  $2^{64}$  different possible values which are scaled to fit the range requested. The randomness comes from the usual Linux/OSX `/dev/urandom` method, and there is no guarantee that it will be truly random, but there is a good chance. It does not show any cycles.

## Performance

*ax* -- a-rate output value.

*kx* -- k-rate output value.

## Examples

Here is an example of the urandom opcode at a-rate. It uses the file `urandom.csd` [examples/urandom.csd].

### Example 1037. An example of the urandom opcode at a-rate.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc    ;;RT audio I/O
```

```

; For Non-realtime ouput leave only the line below:
; -o rnd31.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Create random numbers at a-rate in the range -2 to 2
aur urandom -2, 2

; Use the random numbers to choose a frequency.
afreq = aur * 500 + 100

a1 oscil 30000, afreq, 1
out a1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
e

</CsScore>
</CsoundSynthesizer>

```

Here is an example of the urandom opcode at k-rate. It uses the file *urandom\_krate.csd* [examples/urandom\_krate.csd].

### Example 1038. An example of the urandom opcode at k-rate.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out Audio in
-odac -iadc ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o rnd31_krate.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Create random numbers at k-rate in the range -1 to 1
; with a uniform distribution.
k1 urandom

```

```
    printks "k1=%f\\n", 0.1, k1
  endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
k1=0.229850
k1=-0.077047
k1=-0.199339
k1=-0.620577
k1=-0.119447
k1=-0.596258
k1=0.525800
k1=-0.171583
k1=-0.017196
k1=-0.974613
k1=-0.036276
```

## Credits

Author: John ffitth

New in version 5.13

# urd

urd — A discrete user-defined-distribution random generator that can be used as a function.

## Description

A discrete user-defined-distribution random generator that can be used as a function.

## Syntax

```
aout = urd(ktableNum)
iout = urd(itableNum)
kout = urd(ktableNum)
```

## Initialization

*itableNum* -- number of table containing the random-distribution function. Such table is generated by the user. See GEN40, GEN41, and GEN42. The table length does not need to be a power of 2

## Performance

*ktableNum* -- number of table containing the random-distribution function. Such table is generated by the user. See GEN40, GEN41, and GEN42. The table length does not need to be a power of 2

*urd* is the same opcode as *duserrnd*, but can be used in function fashion.

For a tutorial about random distribution histograms and functions see:

- D. Lorrain. "A panoply of stochastic cannons". In C. Roads, ed. 1989. Music machine. Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the urd opcode. It uses the file *urd.csd* [examples/urd.csd].

### Example 1039. Example of the urd opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o urd.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
```

```
Odbfs = 1

instr 1

ktab = 1 ;ftable 1
kurd = urd(ktab)
ktrig metro 5 ;triggers 5 times per second
kres samphold kurd, ktrig ;sample and hold value of kurd
printk2 kres ;print it
asig poscil .5, 220+kres, 2
outs asig, asig

endin

instr 2

seed 0 ;every run new values

ktab = 1 ;ftable 1
kurd = urd(ktab)
ktrig metro 5 ;triggers 5 times per second
kres samphold kurd, ktrig ;sample and hold value of kurd
printk2 kres ;print it
asig poscil .5, 220+kres, 2
outs asig, asig

endin
</CsInstruments>
<CsScore>
f1 0 -20 -42 10 20 .3 100 200 .7 ;30% choose between 10 and 20 and 70% between 100 and 200
f2 0 8192 10 1 ;sine wave

i 1 0 5
i 2 6 5
e
</CsScore>
</CsSoundSynthesizer>
```

Its output should include lines like these:

```
i1 184.61538
i1 130.76923
i1 169.23077
i1 12.00000
.....

WARNING: Seeding from current time 3751086165

i2 138.46154
i2 12.00000
i2 123.07692
i2 161.53846
i2 123.07692
i2 153.84615
.....
```

## See Also

*cuserrnd, duserrnd*

## Credits

Author: Gabriel Maldonado

New in Version 4.16

# vactrol

vactrol — Envelope follower unit generator.

## Description

Envelope follower unit generator emulating a Perkin Elmer Vactrole VTL5C3/2.

## Syntax

```
ares vactrol asig [iup, idown]
```

## Initialisation

*iup* -- The rise time of the filter, which defaults to 20,

*idown* -- The fall time of the filter, which defaults to 3000,

## Performance

*asig* -- The signal from which to extract the envelope.

## Examples

Here is an example of the vactrol opcode. It uses the file *vactrol.csd* [examples/vactrol.csd].

### Example 1040. Example of the vactrol opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>

<CsInstruments>
  nchnls = 2
  0dbfs = 1

  instr 1
    a1 lfo 0.3, 1, 4
    a2 vactrol a1
    a3 oscili 2, 440
    out a1*a3, a2*a3
  endin
</CsInstruments>

<CsScore>
  i1 0 3
  e
</CsScore>

</CsoundSynthesizer>
```

This opcode can reduce zipper noise produced from complex envelope tracking as it smooths the transitions.

## Credits

Author: John ffitch  
after Julian Parker  
New in 6.04

# vadd

vadd — Adds a scalar value to a vector in a table.

## Description

Adds a scalar value to a vector in a table.

## Syntax

```
vadd ifn, kval, kelements [ , kdstoffset] [ , kverbose]
```

## Initialization

*ifn* - number of the table hosting the vectorial signal to be processed

## Performance

*kval* - scalar value to be added

*kelements* - number of elements of the vector

*kdstoffset* - index offset for the destination table (Optional, default=0)

*kverbose* - Selects whether or not warnings are printed (Optional, default=0)

*vadd* adds the value of *kval* to each element of the vector contained in the table *ifn*, starting from table index *kdstoffset*. This enables you to process a specific section of a table by specifying the offset and the number of elements to be processed. Offset is counted starting from 0, so if no offset is specified (or set to 0), the table will be modified from the beginning.

Note that this opcode runs at k-rate so the value of *kval* is added every control period. Use with care or you will end up with very large numbers (or use *vadd\_i*).

These opcodes (*vadd*, *vmult*, *vpow* and *vexp*) perform numeric operations between a vectorial control signal (hosted by the table *ifn*), and a scalar signal (*kval*). Result is a new vector that overrides old values of *ifn*. All these opcodes work at k-rate.

Negative values for *kdstoffset* are valid. Elements from the vector that are outside the table, will be discarded, and they will not wrap around the table.

If the optional *kverbose* argument is different to 0, the opcode will print warning messages every k-pass if table lengths are exceeded.

In all these opcodes, the resulting vectors are stored in *ifn*, overriding the initial vectors. If you want to keep initial vector, use *vcopy* or *vcopy\_i* to copy it in another table. All these operators are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2*, etc. They can also be useful in conjunction with the spectral opcodes *pvsftw* and *pvsftr*.



### Note

Please note that the *elements* argument has changed in version 5.03 from i-rate to k-rate. This will change the opcode's behavior in the unusual cases where the i-rate variable *ielements* is changed inside the instrument, for example in:

```
instr 1
ielements = 10
vadd 1, 1, ielements
ielements = 20
vadd 2, 1, ielements
turnoff
endin
```

## Examples

Here is an example of the vadd opcode. It uses the file *vadd.csd* [examples/vadd.csd].

### Example 1041. Example of the vadd opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o cigoto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr=44100
ksmps=128
nchnls=2

instr 1
ifn1 = p4
ival = p5
ielements = p6
idstoffset = p7
kval init 25
vadd ifn1, ival, ielements, idstoffset, 1
endin

instr 2 ;Printtable
itable = p4
isize = ftlen(itable)
kcount init 0
kval table kcount, itable
printk2 kval

if (kcount == isize) then
turnoff
endif

kcount = kcount + 1
endin

</CsInstruments>

<CsScore>

f 1 0 16 -7 1 16 17

i2 0.0 0.2 1
```

```
i1 0.4 0.01 1 5 3 4
i2 0.8 0.2 1
i1 1.0 0.01 1 8 5 -3
i2 1.2 0.2 1
i1 1.4 0.01 1 1 10 12
i2 1.6 0.2 1
e
```

```
</CsScore>
```

```
</CsoundSynthesizer>
```

## See also

*vadd\_i*, *vmult*, *vpow* and *vexp*.

## Credits

Written by Gabriel Maldonado. Optional arguments added by Andres Cabrera and Istvan Varga.

New in Csound 5 (Previously available only on CsoundAV)

# vadd\_i

vadd\_i — Adds a scalar value to a vector in a table.

## Description

Adds a scalar value to a vector in a table.

## Syntax

```
vadd_i ifn, ival, ielements [, idstoffset]
```

## Initialization

*ifn* - number of the table hosting the vectorial signal to be processed

*ielements* - number of elements of the vector

*ival* - scalar value to be added

*idstoffset* - index offset for the destination table

## Performance

*vadd\_i* adds the value of *ival* to each element of the vector contained in the table *ifn*, starting from table index *idstoffset*. This enables you to process a specific section of a table by specifying the offset and the number of elements to be processed. Offset is counted starting from 0, so if no offset is specified (or set to 0), the table will be modified from the beginning.

This opcode runs only on initialization, there is a k-rate version of this opcode called *vadd*.

Negative values for *idstoffset* are valid. Elements from the vector that are outside the table, will be discarded, and they will not wrap around the table.

In all these opcodes, the resulting vectors are stored in *ifn*, overriding the initial vectors. If you want to keep initial vector, use *vcopy* or *vcopy\_i* to copy it in another table. All these operators are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2*, etc. They can also be useful in conjunction with the spectral opcodes *pvsftw* and *pvsftr*.

## Examples

Here is an example of the *vadd\_i* opcode. It uses the file *vadd\_i.csd* [examples/vadd\_i.csd].

### Example 1042. Example of the vadd\_i opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform
```

```
; Audio out   Audio in
-odac         -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o cigoto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr=44100
ksmps=128
nchnls=2

instr 1
ifn1 = p4
ival = p5
ielements = p6
idstoffset = p7
kval init 25
vadd_i ifn1, ival, ielements, idstoffset
endin

instr 2 ;Printtable
itable = p4
isize = ftlen(itable)
kcount init 0
kval table kcount, itable
printk2 kval

if (kcount == isize) then
  turnoff
endif

kcount = kcount + 1
endin

</CsInstruments>

<CsScore>

f 1 0 16 -7 1 16 17

i2 0.0 0.2 1
i1 0.4 0.01 1 2 3 4
i2 0.8 0.2 1
i1 1.0 0.01 1 0.5 5 -3
i2 1.2 0.2 1
i1 1.4 0.01 1 1.5 10 12
i2 1.6 0.2 1
e

</CsScore>

</CsoundSynthesizer>
```

## See also

*vadd*, *vmult\_i*, *vpow\_i* and *vexp\_i*.

## Credits

Written by Gabriel Maldonado. Optional arguments added by Andres Cabrera and Istvan Varga.

New in Csound 5 (Previously available only on CsoundAV)

# vaddv

vaddv — Performs addition between two vectorial control signals.

## Description

Performs addition between two vectorial control signals.

## Syntax

```
vaddv ifn1, ifn2, kelements [, kdstoffset] [, ksrcoffset] [,kverbose]
```

## Initialization

*ifn1* - number of the table hosting the first vector to be processed

*ifn2* - number of the table hosting the second vector to be processed

## Performance

*kelements* - number of elements of the two vectors

*kdstoffset* - index offset for the destination (*ifn1*) table (Default=0)

*ksrcoffset* - index offset for the source (*ifn2*) table (Default=0)

*kverbose* - Selects whether or not warnings are printed (Default=0)

*vaddv* adds two vectorial control signals, that is, each element of the first vector is processed (only) with the corresponding element of the other vector. Each vectorial signal is hosted by a table (*ifn1* and *ifn2*). The number of elements contained in both vectors must be the same.

The result is a new vectorial control signal that overrides old values of *ifn1*. If you want to keep the old *ifn1* vector, use *vcopy\_iopcode* to copy it in another table. You can use *kdstoffset* and *ksrcoffset* to specify vectors in any location of the tables.

Negative values for *kdstoffset* and *ksrcoffset* are acceptable. If *kdstoffset* is negative, the out of range section of the vector will be discarded. If *ksrcoffset* is negative, the out of range elements will be assumed to be 0 (i.e. the destination elements will not be changed). If elements for the destination vector are beyond the size of the table (including guard point), these elements are discarded (i.e. elements do not wrap around the tables). If elements for the source vector are beyond the table length, these elements are taken as 0 (i.e. the destination vector will not be changed for these elements).



### Warning

Using the same table as source and destination table in versions earlier than 5.04, might produce unexpected behavior, so use with care.

This opcode works at k-rate (this means that every k-pass the vectors are added). There's an i-rate version of this opcode called *vaddv\_i*.



## Note

Please note that the *elements* argument has changed in version 5.03 from i-rate to k-rate. This will change the opcode's behavior in the unusual cases where the i-rate variable *ielements* is changed inside the instrument, for example in:

```
instr 1
  ielements = 10
            vadd 1, 1, ielements
  ielements = 20
            vadd 2, 1, ielements
            turnoff
endin
```

All these operators (*vaddv*, *vsubv*, *vmultv*, *vdivv*, *vpowv*, *vexpv*, *vcopy* and *vmap*) are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2* etc.

## Examples

Here is an example of the *vaddv* opcode. It uses the file *vaddv.csd* [examples/vaddv.csd].

### Example 1043. Example of the *vaddv* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc          -nm0   ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o cigoto.wav -W   ;; for file output any platform
</CsOptions>
<CsInstruments>

sr=44100
ksmps=128
nchnls=2

opcode TableDumpSimp, 0, ijo
;prints the content of a table in a simple way
ifn, iprec, ippr  xin; function table, float precision while printing (default = 3), parameters per row
iprec = (iprec == -1 ? 3 : iprec)
ippr = (ippr == 0 ? 10 : ippr)
iend = ftlen(ifn)
indx = 0
Sformat sprintf "%%.%df\t", iprec
Sdump = ""
loop:
ival tab_i indx, ifn
Snew sprintf Sformat, ival
Sdump strcat Sdump, Snew
indx = indx + 1
imod = indx % ippr
if imod == 0 then
  puts Sdump, 1
Sdump = ""
endif
if indx < iend igoto loop
```

```

    puts Sdump, 1
endop

instr 1
ifn1 = p4
ifn2 = p5
ielements = p6
idstoffset = p7
isrcoffset = p8
kval init 25
vaddv ifn1, ifn2, ielements, idstoffset, isrcoffset, 1
turnoff
endin

instr 2
TableDumpSimp p4, 3, 16
endin

</CsInstruments>
<CsScore>

f 1 0 16 -7 1 15 16

f 2 0 16 -7 1 15 2

i2 0.0 0.2 1
i2 0.2 0.2 2
i1 0.4 0.01 1 2 5 3 8
i2 0.8 0.2 1
i1 1.0 0.01 1 2 5 10 -2
i2 1.2 0.2 1
i1 1.4 0.01 1 2 8 14 0
i2 1.6 0.2 1
i1 1.8 0.01 1 2 8 0 14
i2 2.0 0.2 1
i1 2.2 0.002 1 1 8 5 2
i2 2.4 0.2 1
e

</CsScore>
</CsoundSynthesizer>

```

## Credits

Written by Gabriel Maldonado. Optional arguments added by Andres Cabrera and Istvan Varga.

New in Csound 5 (Previously available only on CsoundAV)

# vaddv\_i

vaddv\_i — Performs addition between two vectorial control signals at init time.

## Description

Performs addition between two vectorial control signals at init time.

## Syntax

```
vaddv_i ifn1, ifn2, ielements [, idstoffset] [, isrcoffset]
```

## Initialization

*ifn1* - number of the table hosting the first vector to be processed

*ifn2* - number of the table hosting the second vector to be processed

*ielements* - number of elements of the two vectors

*idstoffset* - index offset for the destination (*ifn1*) table (Default=0)

*isrcoffset* - index offset for the source (*ifn2*) table (Default=0)

## Performance

*vaddv\_i* adds two vectorial control signals, that is, each element of the first vector is processed (only) with the corresponding element of the other vector. Each vectorial signal is hosted by a table (*ifn1* and *ifn2*). The number of elements contained in both vectors must be the same.

The result is a new vectorial control signal that overrides old values of *ifn1*. If you want to keep the old *ifn1* vector, use *vcopy\_i* opcode to copy it in another table. You can use *idstoffset* and *isrcoffset* to specify vectors in any location of the tables.

Negative values for *idstoffset* and *isrcoffset* are acceptable. If *idstoffset* is negative, the out of range section of the vector will be discarded. If *isrcoffset* is negative, the out of range elements will be assumed to be 0 (i.e. the destination elements will not be changed). If elements for the destination vector are beyond the size of the table (including guard point), these elements are discarded (i.e. elements do not wrap around the tables). If elements for the source vector are beyond the table length, these elements are taken as 0 (i.e. the destination vector will not be changed for these elements).



### Warning

Using the same table as source and destination table in versions earlier than 5.04, might produce unexpected behavior, so use with care.

This opcode works at init time. There's an k-rate version of this opcode called *vaddv*.

All these operators (*vaddv\_i*, *vsubv\_i*, *vmultv\_i*, *vdivv\_i*, *vpowv\_i*, *vexpv\_i*, *vcopy* and *vmap*) are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2* etc.

## Credits

Written by Gabriel Maldonado. Optional arguments added by Andres Cabrera and Istvan Varga.

New in Csound 5 (Previously available only on CsoundAV)

# vaget

vaget — Access values of the current buffer of an a-rate variable by indexing.

## Description

Access values of the current buffer of an a-rate variable by indexing. Useful for doing sample-by-sample manipulation at k-rate without using setksmps 1.



### Note

Because this opcode does not do any bounds checking, the user must be careful not to try to read values past ksmps (the size of a buffer for an a-rate variable) by using index values greater than ksmps.



### Note

In Csound6 this opcode does do bounds checking.

## Syntax

```
kval vaget kndx, avar
```

## Performance

*kval* - value read from avar

*kndx* - index of the sample to read from the current buffer of the given avar variable

*avar* - a-rate variable to read from

## Examples

Here is an example of the vaget opcode. It uses the file *vaget.csd* [examples/vaget.csd].

### Example 1044. Example of the vaget opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o avarget.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
sr=44100
ksmps=16
nchnls=2
```

```
instr 1 ; Sqrt Signal
ifreq = (p4 > 15 ? p4 : cpspch(p4))
iamp = ampdb(p5)

aout init 0
ksampnum init 0

kenv linseg 0, p3 * .5, 1, p3 * .5, 0

aout1 vco2 1, ifreq
aout2 vco2 .5, ifreq * 2
aout3 vco2 .2, ifreq * 4

aout sum aout1, aout2, aout3

;Take Sqrt of signal, checking for negatives
kcount = 0

loopStart:

kval vaget kcount, aout

if (kval > .0) then
  kval = sqrt(kval)
elseif (kval < 0) then
  kval = sqrt(-kval) * -1
else
  kval = 0
endif

vaset kval, kcount, aout

loop_lt kcount, 1, ksmps, loopStart

aout = aout * kenv

aout moogladder aout, 8000, .1

aout = aout * iamp

outs aout, aout
endin

</CsInstruments>

<CsScore>

i1 0.0 2 440 80
e

</CsScore>

</CsoundSynthesizer>
```

## See Also

*vaset*

## Credits

Author: Steven Yi

New in version 5.04

September 2006.

# valpass

valpass — Variably reverberates an input signal with a flat frequency response.

## Description

Variably reverberates an input signal with a flat frequency response.

## Syntax

```
ares valpass asig, krvt, xlpt, imaxlpt [, iskip] [, insmps]
```

## Initialization

*imaxlpt* -- maximum loop time for *klpt*

*iskip* (optional, default=0) -- initial disposition of delay-loop data space (cf. *reson*). The default value is 0.

*insmps* (optional, default=0) -- delay amount, as a number of samples.

## Performance

*krvt* -- the reverberation time (defined as the time in seconds for a signal to decay to 1/1000, or 60dB down from its original amplitude).

*xlpt* -- variable loop time in seconds, same as *ilpt* in *comb*. Loop time can be as large as *imaxlpt*.

This filter reiterates input with an echo density determined by loop time *xlpt*. The attenuation rate is independent and is determined by *krvt*, the reverberation time (defined as the time in seconds for a signal to decay to 1/1000, or 60dB down from its original amplitude). Its output will begin to appear immediately.

## Examples

Here is an example of the valpass opcode. It uses the file *valpass.csd* [examples/valpass.csd].

### Example 1045. Example of the valpass opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o valpass.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
```

```

ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

krvt = 1.5
klpt line p4, p3, p5
imaxlpt = .1

a1 diskin2 "fox.wav", 1
a1 valpass a1, krvt, klpt, imaxlpt
a2 valpass a1, krvt, klpt*.5, imaxlpt
outs a1, a2

endin
</CsInstruments>
<CsScore>

i 1 0 5 .01 .2
e
</CsScore>
</CsoundSynthesizer>

```

Here is another example of the valpass opcode. It uses the file *valpass-2.csd* [examples/valpass-2.csd].

### Example 1046. Second example of the valpass opcode.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o valpass-2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine ftgen 0, 0, 65536, 10, 1 ;sine wave

instr 1

asig diskin2 "beats.wav", 1, 0, 1
krvt line 0.01, p3, p3 ;reverb time
adepth = p4 ;sine depth
krate = 0.3 ;sine rate (speed)
adel oscil 0.5, krate, giSine ;delay time oscillator (LFO)
adel = ((adel+0.5)*adepth) ;scale and offset LFO
aout valpass asig, krvt, adel*0.01, 0.5
outs aout, aout

endin
</CsInstruments>
<CsScore>

i1 0 10 1
i1 11 10 5
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*alpass, comb, reverb, vcomb*

## Credits

Author: William "Pete" Moss  
University of Texas at Austin  
Austin, Texas USA  
January 2002

# vaset

vaset — Write value of into the current buffer of an a-rate variable by index.

## Description

Write values into the current buffer of an a-rate variable at the given index. Useful for doing sample-by-sample manipulation at k-rate without using setksmps 1.



### Note

Because this opcode does not do any bounds checking, the user must be careful not to try to write values past ksmps (the size of a buffer for an a-rate variable) by using index values greater than ksmps.



### Note

In Csound6 this opcode does do bounds checking.

## Syntax

```
vaset kval, kndx, avar
```

## Performance

*kval* - value to write into avar

*kndx* - index of the sample to write to the current buffer of the given avar variable

*avar* - a-rate variable to write to

## Examples

Here is an example of the vaset opcode. It uses the file *vaset.csd* [examples/vaset.csd].

### Example 1047. Example of the vaset opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o avarset.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
sr=44100
ksmps=1
nchnls=2
```

```
instr 1 ; Sine Wave
ifreq = (p4 > 15 ? p4 : cpspch(p4))
iamp = ampdb(p5)

kenv adsr 0.1, 0.05, .9, 0.2

aout init 0
ksampnum init 0

kcount = 0

iperiod = sr / ifreq
i2pi = 3.14159 * 2

loopStart:

kphase = (ksampnum % iperiod) / iperiod
knewval = sin(kphase * i2pi)

vaset knewval, kcount, aout

ksampnum = ksampnum + 1

loop_lt kcount, 1, ksmps, loopStart

aout = aout * iamp * kenv

outs aout, aout
endin

</CsInstruments>

<CsScore>

i1 0.0 2 440 80
e

</CsScore>

</CsoundSynthesizer>
```

## See Also

*vaget*

## Credits

Author: Steven Yi

New in version 5.04

September 2006.

# vbap

vbap — Distributes an audio signal among many channels.

## Description

Distributes an audio signal among many channels, up to 64 in the first form, arbitrary in the second.

## Syntax

```
ar1[, ar2...] vbap asig, kazim [,  
    kelev] [, kspread] [, ilayout]  
  
array[] vbap asig, kazim [,  
    kelev] [, kspread] [, ilayout]
```

## Initialization

*ilayout* -- index of the speaker layout in the range 0-99, corresponding to a call to *vbaplsinit*.

## Performance

*asig* -- audio signal to be panned

*kazim* -- azimuth angle of the virtual source

*kelev* (optional) -- elevation angle of the virtual source

*kspread* (optional) -- spreading of the virtual source (range 0 - 100). If value is zero, conventional amplitude panning is used. When *kspread* is increased, the number of loudspeakers used in panning increases. If value is 100, the sound is applied to all loudspeakers.

*vbap* takes an input signal, *asig* and distributes it among the outputs, according to the controls *kazim* and *kelev*, and the configured loudspeaker placement. If *idim* = 2, *kelev* is set to zero. The distribution is performed using Vector Base Amplitude Panning (VBAP - See reference). VBAP distributes the signal using loudspeaker data configured with *vbaplsinit*. The signal is applied to, at most, two loudspeakers in 2-D loudspeaker configurations, and three loudspeakers in 3-D loudspeaker configurations. If the virtual source is panned outside the region spanned by loudspeakers, the nearest loudspeakers are used in panning.



### Warning

Please note that all *vbap* panning opcodes require the *vbap* system to be initialized using *vbaplsinit*.

## Examples

See the entry for *vbap8* for an example of usage of the *vbap* opcodes.

## Reference

Ville Pulkki: “Virtual Sound Source Positioning Using Vector Base Amplitude Panning” *Journal of the Audio Engineering Society*, 1997 June, Vol. 45/6, p. 456.

## Examples

Here is an example of the vbap opcode. It uses the file *vbap.csd* [examples/vbap.csd].

### Example 1048. Example of the vbap opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o vbap.wav -W   ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 4 ;quad
0dbfs = 1

vbaplsinit 2, 4, 0, 90, 180, 270

instr 1

asig diskin2 "beats.wav", 1, 0, 1 ;loop beats.wav
kaz line 0, p3, p4 ;come from right rear speaker &
a1,a2,a3,a4 vbap asig, 180, 100, kaz ;change spread of soundsource
printks "spread of source = %d\n", 1, kaz ;print spread value
outq a1,a2,a3,a4

endin
</CsInstruments>
<CsScore>

i 1 0 12 100

e
</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like these:

```

spread of source = 0
spread of source = 8
spread of source = 17
spread of source = 25
spread of source = 33
spread of source = 42
spread of source = 50
spread of source = 58
spread of source = 67
spread of source = 75
spread of source = 83
spread of source = 92
spread of source = 100

```

## See Also

*vbapmove, vbap16, vbap16move, vbap4, vbap4move, vbap8, vbap8move, vbaplsinit, vbapz, vbapzmove*

## Credits

Author: Ville Pulkki  
Sibelius Academy Computer Music Studio  
Laboratory of Acoustics and Audio Signal Processing  
Helsinki University of Technology  
Helsinki, Finland  
May 2000

Author: John ffitc  
July 2012, Sept 2013

New in Csound Version 5.17.13.

Array version new in Csound Version 6.01.

# vbapmove

vbapmove — Distributes an audio signal among many channels with moving virtual sources.

## Description

Distributes an audio signal among upto 64 channels with moving virtual sources.

## Syntax

```
ar1[, ar2...] vbapmove asig, idur, ispread, ifldnum, ifld1 \  
[, ifld2] [...]  
  
aarray[] vbapmove asig, idur, ispread, ifldnum, ifld1 \  
[, ifld2] [...]
```

## Initialization

*idur* -- the duration over which the movement takes place.

*ispread* -- spreading of the virtual source (range 0 - 100). If value is zero, conventional amplitude panning is used. When *ispread* is increased, the number of loudspeakers used in panning increases. If value is 100, the sound is applied to all loudspeakers.

*ifldnum* -- number of fields (absolute value must be 2 or larger). If *ifldnum* is positive, the virtual source movement is a polyline specified by given directions. Each transition is performed in an equal time interval. If *ifldnum* is negative, specified angular velocities are applied to the virtual source during specified relative time intervals (see below).

*ifld1, ifld2, ...* -- azimuth angles or angular velocities, and relative durations of movement phases (see below).

## Performance

*asig* -- audio signal to be panned

*vbapmove* allows the use of moving virtual sources. If *ifldnum* is positive, the fields represent directions of virtual sources and equal times, *iazi1, [iele1,] iazi2, [iele2,]*, etc. The position of the virtual source is interpolated between directions starting from the first direction and ending at the last. Each interval is interpolated in time that is fraction  $\text{total\_time} / \text{number\_of\_intervals}$  of the duration of the sound event.

If *ifldnum* is negative, the fields represent angular velocities and equal times. The first field is, however, the starting direction, *iazi1, [iele1,] iazi\_vel1, [iele\_vel1,] iazi\_vel2, [iele\_vel2,]* .... Each velocity is applied to the note that is fraction  $\text{total\_time} / \text{number\_of\_velocities}$  of the duration of the sound event. If the elevation of the virtual source becomes greater than 90 degrees or less than 0 degrees, the polarity of angular velocity is changed. Thus the elevational angular velocity produces a virtual source that moves up and down between 0 and 90 degrees.



### Warning

Please note that all *vbap* panning opcodes require the *vbap* system to be initialized using *vbaplsinit*.

## Examples

See the entry for *vbap8move* for an example of usage of the *vbapXmove* opcodes.

## Reference

Ville Pulkki: “Virtual Sound Source Positioning Using Vector Base Amplitude Panning” *Journal of the Audio Engineering Society*, 1997 June, Vol. 45/6, p. 456.

## Examples

Here is an example of the *vbapmove* opcode. It uses the file *vbapmove.csd* [examples/vbapmove.csd].

### Example 1049. Example of the *vbapmove* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o vbapmove.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 4 ;quad
0dbfs = 1

vbaplsinit 2, 4, 0, 90, 180, 270

instr 1

asig diskin2 "beats.wav", 1, 0, 1 ;loop beats.wav
a1,a2,a3,a4 vbapmove asig, p3, 1, 2, 310, 180 ;change movement of soundsource in
      outq a1,a2,a3,a4 ;the rear speakers

endin
</CsInstruments>
<CsScore>

i 1 0 5

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*vbap*, *vbap16*, *vbap16move*, *vbap4*, *vbap4move*, *vbap8*, *vbap8move*, *vbaplsinit*, *vbapz*, *vbapzmove*

## Credits

Author: Ville Pulkki

Sibelius Academy Computer Music Studio  
Laboratory of Acoustics and Audio Signal Processing  
Helsinki University of Technology  
Helsinki, Finland  
May 2000  
Author: John fitch  
July 2012, Sept 2013

New in Csound Version 5.17.13

Array outputs new in Csound Version 6.01

# vbapg

vbapg — Calculates the gains for a sound location between multiple channels.

## Description

Calculates the gains for a sound location for up to 64.

## Syntax

```
k1[, k2...] vbapg kazim [,kelev] [, kspread] [, ilayout]
karray[] vbapg kazim [,kelev] [, kspread] [, ilayout]
```

## Initialization

*ilayout* -- index of the speaker layout in the range 0-99, corresponding to a call to *vbaplsinit*. The default value is 0.

## Performance

*kazim* -- azimuth angle of the virtual source

*kelev* (optional) -- elevation angle of the virtual source

*kspread* (optional) -- spreading of the virtual source (range 0 - 100). If value is zero, conventional amplitude panning is used. When *kspread* is increased, the number of loudspeakers used in panning increases. If value is 100, the sound is applied to all loudspeakers.

*vbapg* calculates the gains that an input signal would have between multiple speakers according to the controls *kazim* and *kelev*, and the configured loudspeaker placement *ilayout*. If *idim* = 2, *kelev* is set to zero. The distribution is performed using Vector Base Amplitude Panning (VBAP - See reference). VBAP distributes the signal using loudspeaker data configured with *vbaplsinit*. The signal is applied to, at most, two loudspeakers in 2-D loudspeaker configurations, and three loudspeakers in 3-D loudspeaker configurations. If the virtual source is panned outside the region spanned by loudspeakers, the nearest loudspeakers are used in panning.



### Warning

Please note that all *vbap* panning opcodes require the *vbap* system to be initialized using *vbaplsinit*.

## Examples

## Reference

Ville Pulkki: “Virtual Sound Source Positioning Using Vector Base Amplitude Panning” *Journal of the Audio Engineering Society*, 1997 June, Vol. 45/6, p. 456.

## Examples

Here is an example of the *vbapg* opcode. It uses the file *vbapg.csd* [examples/vbapg.csd].

**Example 1050. Example of the vbapg opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o vbap4.wav -W   ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 4 ;quad
0dbfs = 1

vbaplsinit 2.01, 4, 0, 90, 180, 270

instr 1

asig disk2 "beats.wav", 1, 0, 1 ;loop beats.wav
kaz line 0, p3, p4 ;come from right rear speaker &
k1,k2,k3,k4 vbapg 180, 100, kaz, 1 ;change spread of soundsource
printks "spread of source = %d\n", 1, kaz ;print spread value
outq asig*k1,asig*k2,asig*k3,asig*k4

endin
</CsInstruments>
<CsScore>

i 1 0 12 100

e
</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like these:

```

spread of source = 0
spread of source = 8
spread of source = 17
spread of source = 25
spread of source = 33
spread of source = 42
spread of source = 50
spread of source = 58
spread of source = 67
spread of source = 75
spread of source = 83
spread of source = 92
spread of source = 100

```

**See Also**

*vbap*, *vbapmove*, *vbap4*, *vbap4move*, *vbap8*, *vbap8move*, *vbap16*, *vbap16move*, *vbaplsinit*, *vbapz*, *vbapz-move*

## Credits

Author: Ville Pulkki  
Sibelius Academy Computer Music Studio  
Laboratory of Acoustics and Audio Signal Processing  
Helsinki University of Technology  
Helsinki, Finland  
May 2000  
Author: John fitch  
July 2012, Sept 2013

New in Csound Version 5.17.13.

Array version new in Csound Version 6.01.

# vbapgmove

`vbapgmove` — Calculates the gains for a sound location between multiple channels with moving virtual sources.

## Description

Calculates the gains for a sound location between multiple channels with moving virtual sources.

## Syntax

```
kr1[, kr2...] vbapgmove idur, ispread, ifldnum, ifld1 \  
[, ifld2] [...]  
  
karray[] vbapgmove idur, ispread, ifldnum, ifld1 \  
[, ifld2] [...]
```

## Initialization

*idur* -- the duration over which the movement takes place.

*ispread* -- spreading of the virtual source (range 0 - 100). If value is zero, conventional amplitude panning is used. When *ispread* is increased, the number of loudspeakers used in panning increases. If value is 100, the sound is applied to all loudspeakers.

*ifldnum* -- number of fields (absolute value must be 2 or larger). If *ifldnum* is positive, the virtual source movement is a polyline specified by given directions. Each transition is performed in an equal time interval. If *ifldnum* is negative, specified angular velocities are applied to the virtual source during specified relative time intervals (see below).

*ifld1*, *ifld2*, ... -- azimuth angles or angular velocities, and relative durations of movement phases (see below).

## Performance

*vbapgmove* allows the use of moving virtual sources. If *ifldnum* is positive, the fields represent directions of virtual sources and equal times, *iazi1*, [*iele1*,] *iazi2*, [*iele2*,], etc. The position of the virtual source is interpolated between directions starting from the first direction and ending at the last. Each interval is interpolated in time that is fraction  $\text{total\_time} / \text{number\_of\_intervals}$  of the duration of the sound event.

If *ifldnum* is negative, the fields represent angular velocities and equal times. The first field is, however, the starting direction, *iazi1*, [*iele1*,] *iazi\_vel1*, [*iele\_vel1*,] *iazi\_vel2*, [*iele\_vel2*,] .... Each velocity is applied to the note that is fraction  $\text{total\_time} / \text{number\_of\_velocities}$  of the duration of the sound event. If the elevation of the virtual source becomes greater than 90 degrees or less than 0 degrees, the polarity of angular velocity is changed. Thus the elevational angular velocity produces a virtual source that moves up and down between 0 and 90 degrees.



### Warning

Please note that all *vbap* panning opcodes require the *vbap* system to be initialized using *vbaplsinit*.

## Examples

Here is an example of the `vbapgmmove` opcode. It uses the file `vbapgmmove.csd` [examples/vbapgmmove.csd].

### Example 1051. Example of the `vbapg` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o vbapgmmove.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 4 ;quad
0dbfs = 1

vbaplsinit 2, 4, 0, 90, 180, 270

instr 1

asig diskin2 "beats.wav", 1, 0, 1 ;loop beats.wav

k0,k1,k2,k3 vbapgmmove p3, 1, 2, 310, 180 ;change movement of soundsource in
outq k0*asig,k1*asig,k2*asig,k3*asig ;the rear speakers

endin
</CsInstruments>
<CsScore>

i 1 0 5

e
</CsScore>
</CsoundSynthesizer>
```

## Reference

Ville Pulkki: “Virtual Sound Source Positioning Using Vector Base Amplitude Panning” *Journal of the Audio Engineering Society*, 1997 June, Vol. 45/6, p. 456.

## See Also

`vbap`, `vbap16`, `vbap16move`, `vbap4`, `vbap4move`, `vbap8`, `vbap8move`, `vbaplsinit`, `vbapz`, `vbapzmove`

## Credits

Author: Ville Pulkki  
Sibelius Academy Computer Music Studio  
Laboratory of Acoustics and Audio Signal Processing  
Helsinki University of Technology

Helsinki, Finland  
May 2000  
Author: John fitch  
July 2012, Sept 2013

New in Csound Version 5.17.13

Array outputs new in Csound Version 6.01

# vbap16

`vbap16` — Distributes an audio signal among 16 channels.

## Description

Distributes an audio signal among 16 channels.

## Syntax

```
ar1, ..., ar16 vbap16 asig, kazim [, kelev] [, kspread]
```

## Performance

*asig* -- audio signal to be panned

*kazim* -- azimuth angle of the virtual source

*kelev* (optional) -- elevation angle of the virtual source

*kspread* (optional) -- spreading of the virtual source (range 0 - 100). If value is zero, conventional amplitude panning is used. When *kspread* is increased, the number of loudspeakers used in panning increases. If value is 100, the sound is applied to all loudspeakers.

*vbap16* takes an input signal, *asig*, and distribute it among 16 outputs, according to the controls *kazim* and *kelev*, and the configured loudspeaker placement. If *idim* = 2, *kelev* is set to zero. The distribution is performed using Vector Base Amplitude Panning (VBAP - See reference). VBAP distributes the signal using loudspeaker data configured with *vbaplsinit*. The signal is applied to, at most, two loudspeakers in 2-D loudspeaker configurations, and three loudspeakers in 3-D loudspeaker configurations. If the virtual source is panned outside the region spanned by loudspeakers, the nearest loudspeakers are used in panning.



### Warning

Please note that all *vbap* panning opcodes require the *vbap* system to be initialized using *vbaplsinit*.

## Examples

See the entry for *vbap8* for an example of usage of the *vbap* opcodes.

## Reference

Ville Pulkki: “Virtual Sound Source Positioning Using Vector Base Amplitude Panning” *Journal of the Audio Engineering Society*, 1997 June, Vol. 45/6, p. 456.

## See Also

*vbap16move*, *vbap4*, *vbap4move*, *vbap8*, *vbap8move*, *vbaplsinit*, *vbapz*, *vbapzmove*

## Credits

Author: Ville Pulkki

Sibelius Academy Computer Music Studio  
Laboratory of Acoustics and Audio Signal Processing  
Helsinki University of Technology  
Helsinki, Finland  
May 2000

New in Csound Version 4.07. Input parameters accept k-rate since Csound 5.09.

# vbap16move

vbap16move — Distribute an audio signal among 16 channels with moving virtual sources.

## Description

Distribute an audio signal among 16 channels with moving virtual sources.

## Syntax

```
ar1, ..., ar16 vbap16move asig, idur, ispread, ifldnum, ifld1 \
[, ifld2] [...]
```

## Initialization

*idur* -- the duration over which the movement takes place.

*ispread* -- spreading of the virtual source (range 0 - 100). If value is zero, conventional amplitude panning is used. When *ispread* is increased, the number of loudspeakers used in panning increases. If value is 100, the sound is applied to all loudspeakers.

*ifldnum* -- number of fields (absolute value must be 2 or larger). If *ifldnum* is positive, the virtual source movement is a polyline specified by given directions. Each transition is performed in an equal time interval. If *ifldnum* is negative, specified angular velocities are applied to the virtual source during specified relative time intervals (see below).

*ifld1, ifld2, ...* -- azimuth angles or angular velocities, and relative durations of movement phases.

## Performance

*asig* -- audio signal to be panned

*vbap16move* allows the use of moving virtual sources. If *ifldnum* is positive, the fields represent directions of virtual sources and equal times, *iazi1, [iele1,] iazi2, [iele2,]*, etc. The position of the virtual source is interpolated between directions starting from the first direction and ending at the last. Each interval is interpolated in time that is fraction  $\text{total\_time} / \text{number\_of\_intervals}$  of the duration of the sound event.

If *ifldnum* is negative, the fields represent angular velocities and equal times. The first field is, however, the starting direction, *iazi1, [iele1,] iazi\_vel1, [iele\_vel1,] iazi\_vel2, [iele\_vel2,]* ... Each velocity is applied to the note that is fraction  $\text{total\_time} / \text{number\_of\_velocities}$  of the duration of the sound event. If the elevation of the virtual source becomes greater than 90 degrees or less than 0 degrees, the polarity of angular velocity is changed. Thus the elevational angular velocity produces a virtual source that moves up and down between 0 and 90 degrees.



### Warning

Please note that all *vbap* panning opcodes require the *vbap* system to be initialized using *vbaplsinit*.

## Examples

See the entry for *vbap8move* for an example of usage of the *vbapXmove* opcodes.

## Reference

Ville Pulkki: “Virtual Sound Source Positioning Using Vector Base Amplitude Panning” *Journal of the Audio Engineering Society*, 1997 June, Vol. 45/6, p. 456.

## See Also

*vbap16, vbap4, vbap4move, vbap8, vbap8move, vbaplsinit, vbapz, vbapzmove, vbapzmove*

## Credits

Author: Ville Pulkki  
Sibelius Academy Computer Music Studio  
Laboratory of Acoustics and Audio Signal Processing  
Helsinki University of Technology  
Helsinki, Finland  
May 2000

New in Csound Version 4.07

# vbap4

vbap4 — Distributes an audio signal among 4 channels.

## Description

Distributes an audio signal among 4 channels.

## Syntax

```
ar1, ar2, ar3, ar4 vbap4 asig, kazim [, kelev] [, kspread]
```

## Performance

*asig* -- audio signal to be panned

*kazim* -- azimuth angle of the virtual source

*kelev* (optional) -- elevation angle of the virtual source

*kspread* (optional) -- spreading of the virtual source (range 0 - 100). If value is zero, conventional amplitude panning is used. When *kspread* is increased, the number of loudspeakers used in panning increases. If value is 100, the sound is applied to all loudspeakers.

*vbap4* takes an input signal, *asig* and distributes it among 4 outputs, according to the controls *kazim* and *kelev*, and the configured loudspeaker placement. If *idim* = 2, *kelev* is set to zero. The distribution is performed using Vector Base Amplitude Panning (VBAP - See reference). VBAP distributes the signal using loudspeaker data configured with *vbaplsinit*. The signal is applied to, at most, two loudspeakers in 2-D loudspeaker configurations, and three loudspeakers in 3-D loudspeaker configurations. If the virtual source is panned outside the region spanned by loudspeakers, the nearest loudspeakers are used in panning.



### Warning

Please note that all *vbap* panning opcodes require the *vbap* system to be initialized using *vbaplsinit*.

## Examples

See the entry for *vbap8* for an example of usage of the *vbap* opcodes.

## Reference

Ville Pulkki: “Virtual Sound Source Positioning Using Vector Base Amplitude Panning” *Journal of the Audio Engineering Society*, 1997 June, Vol. 45/6, p. 456.

## Examples

Here is an example of the *vbap4* opcode. It uses the file *vbap4.csd* [examples/vbap4.csd].

### Example 1052. Example of the *vbap4* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o vbap4.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 4 ;quad
0dbfs = 1

vbaplsinit 2, 4, 0, 90, 180, 270

instr 1

asig diskin2 "beats.wav", 1, 0, 1 ;loop beats.wav
kaz line 0, p3, p4 ;come from right rear speaker &
a1,a2,a3,a4 vbap4 asig, 180, 100, kaz ;change spread of soundsource
printks "spread of source = %d\n", 1, kaz ;print spread value
outq a1,a2,a3,a4

endin
</CsInstruments>
<CsScore>

i 1 0 12 100

e
</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like these:

```
spread of source = 0
spread of source = 8
spread of source = 17
spread of source = 25
spread of source = 33
spread of source = 42
spread of source = 50
spread of source = 58
spread of source = 67
spread of source = 75
spread of source = 83
spread of source = 92
spread of source = 100
```

## See Also

*vbap16, vbap16move, vbap4move, vbap8, vbap8move, vbaplsinit, vbapz, vbapzmove*

## Credits

Author: Ville Pulkki  
Sibelius Academy Computer Music Studio  
Laboratory of Acoustics and Audio Signal Processing

Helsinki University of Technology  
Helsinki, Finland  
May 2000

New in Csound Version 4.06. Input parameters accept k-rate since Csound 5.09.

# vbap4move

vbap4move — Distributes an audio signal among 4 channels with moving virtual sources.

## Description

Distributes an audio signal among 4 channels with moving virtual sources.

## Syntax

```
ar1, ar2, ar3, ar4 vbap4move asig, idur, ispread, ifldnum, ifld1 \  
[, ifld2] [...]
```

## Initialization

*idur* -- the duration over which the movement takes place.

*ispread* -- spreading of the virtual source (range 0 - 100). If value is zero, conventional amplitude panning is used. When *ispread* is increased, the number of loudspeakers used in panning increases. If value is 100, the sound is applied to all loudspeakers.

*ifldnum* -- number of fields (absolute value must be 2 or larger). If *ifldnum* is positive, the virtual source movement is a polyline specified by given directions. Each transition is performed in an equal time interval. If *ifldnum* is negative, specified angular velocities are applied to the virtual source during specified relative time intervals (see below).

*ifld1, ifld2, ...* -- azimuth angles or angular velocities, and relative durations of movement phases (see below).

## Performance

*asig* -- audio signal to be panned

*vbap4move* allows the use of moving virtual sources. If *ifldnum* is positive, the fields represent directions of virtual sources and equal times, *iazi1, [iele1,] iazi2, [iele2,]*, etc. The position of the virtual source is interpolated between directions starting from the first direction and ending at the last. Each interval is interpolated in time that is fraction  $\text{total\_time} / \text{number\_of\_intervals}$  of the duration of the sound event.

If *ifldnum* is negative, the fields represent angular velocities and equal times. The first field is, however, the starting direction, *iazi1, [iele1,] iazi\_vel1, [iele\_vel1,] iazi\_vel2, [iele\_vel2,]* .... Each velocity is applied to the note that is fraction  $\text{total\_time} / \text{number\_of\_velocities}$  of the duration of the sound event. If the elevation of the virtual source becomes greater than 90 degrees or less than 0 degrees, the polarity of angular velocity is changed. Thus the elevational angular velocity produces a virtual source that moves up and down between 0 and 90 degrees.



### Warning

Please note that all *vbap* panning opcodes require the *vbap* system to be initialized using *vbaplsinit*.

## Examples

See the entry for *vbap8move* for an example of usage of the *vbapXmove* opcodes.

## Reference

Ville Pulkki: “Virtual Sound Source Positioning Using Vector Base Amplitude Panning” *Journal of the Audio Engineering Society*, 1997 June, Vol. 45/6, p. 456.

## Examples

Here is an example of the `vbap4move` opcode. It uses the file `vbap4move.csd` [examples/vbap4move.csd].

### Example 1053. Example of the `vbap4move` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o vbap4move.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 4 ;quad
0dbfs = 1

vbaplsinit 2, 4, 0, 90, 180, 270

instr 1

asig disk12 "beats.wav", 1, 0, 1 ;loop beats.wav
a1,a2,a3,a4 vbap4move asig, p3, 1, 2, 310, 180 ;change movement of soundsource in
          outq a1,a2,a3,a4 ;the rear speakers

endin
</CsInstruments>
<CsScore>

i 1 0 5

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

`vbap16`, `vbap16move`, `vbap4`, `vbap8`, `vbap8move`, `vbaplsinit`, `vbapz`, `vbapzmove`

## Credits

Author: Ville Pulkki  
Sibelius Academy Computer Music Studio  
Laboratory of Acoustics and Audio Signal Processing  
Helsinki University of Technology  
Helsinki, Finland

May 2000

New in Csound Version 4.07

# vbap8

vbap8 — Distributes an audio signal among 8 channels.

## Description

Distributes an audio signal among 8 channels.

## Syntax

```
ar1, ..., ar8 vbap8 asig, kazim [, kelev] [, kspread]
```

## Performance

*asig* -- audio signal to be panned

*kazim* -- azimuth angle of the virtual source

*kelev* (optional) -- elevation angle of the virtual source

*kspread* (optional) -- spreading of the virtual source (range 0 - 100). If value is zero, conventional amplitude panning is used. When *kspread* is increased, the number of loudspeakers used in panning increases. If value is 100, the sound is applied to all loudspeakers.

*vbap8* takes an input signal, *asig*, and distributes it among 8 outputs, according to the controls *kazim* and *kelev*, and the configured loudspeaker placement. If *idim* = 2, *kelev* is set to zero. The distribution is performed using Vector Base Amplitude Panning (VBAP - See reference). VBAP distributes the signal using loudspeaker data configured with *vbaplsinit*. The signal is applied to, at most, two loudspeakers in 2-D loudspeaker configurations, and three loudspeakers in 3-D loudspeaker configurations. If the virtual source is panned outside the region spanned by loudspeakers, the nearest loudspeakers are used in panning.



### Warning

Please note that all *vbap* panning opcodes require the *vbap* system to be initialized using *vbaplsinit*.

## Example

Here is a simple example of the *vbap8* opcode. It uses the file *vbap8.csd* [examples/vbap8.csd].

### Example 1054. Example of the vbap8 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
;-odac      -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
-o vbap8.wav -W ;; for file output any platform
```

```
</CsOptions>
<CsInstruments>

  sr      =      44100
  kr      =      441
  ksmps   =      100
  nchnls  =      4
  vbaplsinit 2, 8, 0, 45, 90, 135, 200, 245, 290, 315

  instr 1
  asig   oscil      20000, 440, 1
  a1,a2,a3,a4,a5,a6,a7,a8  vbap8  asig, p4, 0, 20 ;p4 = azimuth

  ;render twice with alternate outq statements
  ; to obtain two 4 channel .wav files:

      outq      a1,a2,a3,a4
  ;      outq      a5,a6,a7,a8
  ; or use an 8-channel output for realtime output (set nchnls to 8):
  ;      outo a1,a2,a3,a4,a5,a6,a7,a8
      endin

</CsInstruments>
<CsScore>
f 1 0 8192 10 1
; Play Instrument #1 for one second.
;      azimuth
i 1 0 1      20
i 1 + .      40
i 1 + .      60
i 1 + .      80
i 1 + .     100
i 1 + .     120
i 1 + .     140
i 1 + .     160
e

</CsScore>
</CsoundSynthesizer>
```

## Reference

Ville Pulkki: “Virtual Sound Source Positioning Using Vector Base Amplitude Panning” *Journal of the Audio Engineering Society*, 1997 June, Vol. 45/6, p. 456.

## See Also

*vbap16, vbap16move, vbap4, vbap4move, vbap8move, vbaplsinit, vbapz, vbapzmove*

## Credits

Author: Ville Pulkki  
Sibelius Academy Computer Music Studio  
Laboratory of Acoustics and Audio Signal Processing  
Helsinki University of Technology  
Helsinki, Finland  
May 2000

New in Csound Version 4.07. Input parameters accept k-rate since Csound 5.09.

# vbap8move

vbap8move — Distributes an audio signal among 8 channels with moving virtual sources.

## Description

Distributes an audio signal among 8 channels with moving virtual sources.

## Syntax

```
ar1, ..., ar8 vbap8move asig, idur, ispread, ifldnum, ifld1 \  
    [, ifld2] [...]
```

## Initialization

*idur* -- the duration over which the movement takes place.

*ispread* -- spreading of the virtual source (range 0 - 100). If value is zero, conventional amplitude panning is used. When *ispread* is increased, the number of loudspeakers used in panning increases. If value is 100, the sound is applied to all loudspeakers.

*ifldnum* -- number of fields (absolute value must be 2 or larger). If *ifldnum* is positive, the virtual source movement is a polyline specified by given directions. Each transition is performed in an equal time interval. If *ifldnum* is negative, specified angular velocities are applied to the virtual source during specified relative time intervals (see below).

*ifld1, ifld2, ...* -- azimuth angles or angular velocities, and relative durations of movement phases (see below).

## Performance

*asig* -- audio signal to be panned

*vbap8move* allows the use of moving virtual sources. If *ifldnum* is positive, the fields represent directions of virtual sources and equal times, *iazi1, [iele1,] iazi2, [iele2,]*, etc. The position of the virtual source is interpolated between directions starting from the first direction and ending at the last. Each interval is interpolated in time that is fraction  $\text{total\_time} / \text{number\_of\_intervals}$  of the duration of the sound event.

If *ifldnum* is negative, the fields represent angular velocities and equal times. The first field is, however, the starting direction, *iazi1, [iele1,] iazi\_vel1, [iele\_vel1,] iazi\_vel2, [iele\_vel2,]* ... Each velocity is applied to the note that is fraction  $\text{total\_time} / \text{number\_of\_velocities}$  of the duration of the sound event. If the elevation of the virtual source becomes greater than 90 degrees or less than 0 degrees, the polarity of angular velocity is changed. Thus the elevational angular velocity produces a virtual source that moves up and down between 0 and 90 degrees.



### Warning

Please note that all *vbap* panning opcodes require the *vbap* system to be initialized using *vbaplsinit*.

## Example

Here is a simple example of the `vbap8move` opcode. It uses the file `vbap8move.csd` [examples/vbap8move.csd].

### Example 1055. Example of the `vbap8move` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc    ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o vbap4move.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 48000
ksmps = 10
nchnls = 8

;Example by Hector Centeno 2007

vbaplsinit    2, 8, 15, 65, 115, 165, 195, 245, 295, 345

    instr 1
ifldnum = 9
ispread = 30
idur = p3

;; Generate a sound source
kenv loopseg 10, 0, 0, 0, 0.5, 1, 10, 0
a1 pinkish 3000*kenv

;; Move circling around once all the speakers
aout1, aout2, aout3, aout4, aout5, aout6, aout7, aout8 vbap8move a1, idur, ispread, ifldnum, 15, 65, 115, 165, 195, 245, 295, 345

;; Speaker mapping
aFL = aout8 ; Front Left
aFR = aout1 ; Front Right
aMFL = aout7 ; Mid Front Left
aMFR = aout2 ; Mid Front Right
aMBL = aout6 ; Mid Back Left
aMBR = aout3 ; Mid Back Right
aBL = aout5 ; Back Left
aBR = aout4 ; Back Right

outo aFL, aFR, aMFL, aMFR, aMBL, aMBR, aBL, aBR

    endin

</CsInstruments>
<CsScore>
i1 0 30
e
</CsScore>
</CsoundSynthesizer>

```

## Reference

Ville Pulkki: “Virtual Sound Source Positioning Using Vector Base Amplitude Panning” *Journal of the Audio Engineering Society*, 1997 June, Vol. 45/6, p. 456.

## See Also

*vbap16, vbap16move, vbap4, vbap4move, vbap8, vbaplsinit, vbapz, vbapzmove*

## Credits

Author: Ville Pulkki  
Sibelius Academy Computer Music Studio  
Laboratory of Acoustics and Audio Signal Processing  
Helsinki University of Technology  
Helsinki, Finland  
May 2000

New in Csound Version 4.07

# vbaplsinit

vbaplsinit — Configures VBAP output according to loudspeaker parameters.

## Description

Configures VBAP output according to loudspeaker parameters.

## Syntax

```
vbaplsinit idim, ilsnum [, idir1] [, idir2] [...] [, idir32]
```

```
vbaplsinit idim, ilsnum, ilsarray
```

## Initialization

*idim* -- dimensionality of loudspeaker array. Either 2 or 3. If the dimension has a fractional part then that is used as the index of the layout created (used in *vbap*, *vbapz* and *vbapg* only). The fractional part should be between .00 and .99.

*ilsnum* -- number of loudspeakers. In two dimensions, the number can vary from 2 to 64. In three dimensions, the number can vary from 3 and 64.

*idir1*, *idir2*, ..., *idir32* -- directions of loudspeakers. Number of directions must be less than or equal to 16. In two-dimensional loudspeaker positioning, *idir<sub>n</sub>* is the azimuth angle respective to *n*th channel. In three-dimensional loudspeaker positioning, fields are the azimuth and elevation angles of each loudspeaker consequently (*azi1*, *ele1*, *azi2*, *ele2*, etc.).

*ilsarray* -- a one dimensional array of the data as described above.



### Note

In 2 dimensions the angle between two adjacent speakers must be less than 179 degrees (170 degrees in earlier versions). This is a restriction of the algorithm.

## Performance

VBAP distributes the signal using loudspeaker data configured with *vbaplsinit*. The signal is applied to, at most, two loudspeakers in 2-D loudspeaker configurations, and three loudspeakers in 3-D loudspeaker configurations. If the virtual source is panned outside the region spanned by loudspeakers, the nearest loudspeakers are used in panning.

## Examples

Here is an example of the *vbaplsinit* opcode. It uses the file *vbaplsinit.csd* [examples/vbaplsinit.csd].

### Example 1056. Example of the *vbaplsinit* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o vbaplsinit.wav -W   ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 8
0dbfs = 1

vbaplsinit 2, 8, 0, 45, 90, 180, 270, 0, 0, 0 ;5 speakers for 5.1 amps

instr 1

asig diskin2 "beats.wav", 1, 0, 1 ;loop beats.wav
kazim line 1, p3, 355
a1,a2,a3,a4,a5,a6,a7,a8 vbap8 asig, kazim, 0, 1 ;change azimuth of soundsource
; Speaker mapping
aFL = a1 ; Front Left
aMF = a5 ; Mid Front
aFR = a2 ; Front Right
aBL = a3 ; Back Left
aBR = a4 ; Back Right
outo aFL,aFR,aBL,aBR,aMF,a6,a7,a8 ;a6, a7 and a8 are dummies

endin
</CsInstruments>
<CsScore>

i 1 0 5

e
</CsScore>
</CsoundSynthesizer>
```

See the other entries of the vbap opcodes for different examples.

## Reference

Ville Pulkki: "Virtual Sound Source Positioning Using Vector Base Amplitude Panning" *Journal of the Audio Engineering Society*, 1997 June, Vol. 45/6, p. 456.

## See Also

*vbap, vbapg, vbap4, vbap4move, vbap8, vbap8move, vbap16, vbap16move, vbapz, vbapzmove*

## Credits

Author: Ville Pulkki  
Sibelius Academy Computer Music Studio  
Laboratory of Acoustics and Audio Signal Processing  
Helsinki University of Technology  
Helsinki, Finland  
May 2000

New in Csound Version 4.07

Multiple layouts new ib version 5.17.14

# vbapz

vbapz — Writes a multi-channel audio signal to a ZAK array.

## Description

Writes a multi-channel audio signal to a ZAK array.

## Syntax

```
vbapz inumchnls, istartndx, asig, kazim [, kelev] [, kspread]
```

## Initialization

*inumchnls* -- number of channels to write to the ZA array. Must be in the range 2 - 256.

*istartndx* -- first index or position in the ZA array to use

## Performance

*asig* -- audio signal to be panned

*kazim* -- azimuth angle of the virtual source

*kelev* (optional) -- elevation angle of the virtual source

*kspread* (optional) -- spreading of the virtual source (range 0 - 100). If value is zero, conventional amplitude panning is used. When *kspread* is increased, the number of loudspeakers used in panning increases. If value is 100, the sound is applied to all loudspeakers.

The opcode *vbapz* is the multiple channel analog of the opcodes like *vbap4*, working on *inumchnls* and using a ZAK array for output.



### Warning

Please note that all *vbap* panning opcodes require the *vbap* system to be initialized using *vbaplsinit*.

## Examples

See the entry for *vbap8* for an example of usage of the *vbap* opcodes.

## Reference

Ville Pulkki: “Virtual Sound Source Positioning Using Vector Base Amplitude Panning” *Journal of the Audio Engineering Society*, 1997 June, Vol. 45/6, p. 456.

## See Also

*vbap16*, *vbap16move*, *vbap4*, *vbap4move*, *vbap8*, *vbap8move*, *vbaplsinit*, *vbapzmove*

## Credits

John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2000

New in Csound Version 4.07. Input parameters accept k-rate since Csound 5.09.

# vbapzmove

vbapzmove — Writes a multi-channel audio signal to a ZAK array with moving virtual sources.

## Description

Writes a multi-channel audio signal to a ZAK array with moving virtual sources.

## Syntax

```
vbapzmove inumchnls, istartndx, asig, idur, ispread, ifldnum, ifld1, \  
ifld2, [...]
```

## Initialization

*inumchnls* -- number of channels to write to the ZA array. Must be in the range 2 - 256.

*istartndx* -- first index or position in the ZA array to use

*idur* -- the duration over which the movement takes place.

*ispread* -- spreading of the virtual source (range 0 - 100). If value is zero, conventional amplitude panning is used. When *ispread* is increased, the number of loudspeakers used in panning increases. If value is 100, the sound is applied to all loudspeakers.

*ifldnum* -- number of fields (absolute value must be 2 or larger). If *ifldnum* is positive, the virtual source movement is a polyline specified by given directions. Each transition is performed in an equal time interval. If *ifldnum* is negative, specified angular velocities are applied to the virtual source during specified relative time intervals (see below).

*ifld1*, *ifld2*, ... -- azimuth angles or angular velocities, and relative durations of movement phases (see below).

## Performance

*asig* -- audio signal to be panned

The opcode *vbapzmove* is the multiple channel analog of the opcodes like *vbap4move*, working on *inumchnls* and using a ZAK array for output.



### Warning

Please note that all *vbap* panning opcodes require the *vbap* system to be initialized using *vbaplsinit*.

## Examples

See the entry for *vbap8move* for an example of usage of the *vbapXmove* opcodes.

## Reference

Ville Pulkki: “Virtual Sound Source Positioning Using Vector Base Amplitude Panning” *Journal of the Audio Engineering Society*, 1997 June, Vol. 45/6, p. 456.

## See Also

*vbap16, vbap16move, vbap4, vbap4move, vbap8, vbap8move, vbaplsinit, vbapz,*

## Credits

John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2000

New in Csound Version 4.07

# vcella

vcella — Cellular Automata

## Description

Unidimensional Cellular Automata applied to Csound vectors

## Syntax

```
vcella ktrig, kreinit, ioutFunc, initStateFunc, \  
      iRuleFunc, ielements, irulelen [, iradius]
```

## Initialization

*ioutFunc* - number of the table where the state of each cell is stored

*initStateFunc* - number of a table containig the inital states of each cell

*iRuleFunc* - number of a lookup table containing the rules

*ielements* - total number of cells

*irulelen* - total number of rules

*iradius* (optional) - radius of Cellular Automata. At present time CA radius can be 1 or 2 (1 is the default)

## Performance

*ktrig* - trigger signal. Each time it is non-zero, a new generation of cells is evaluated

*kreinit* - trigger signal. Each time it is non-zero, state of all cells is forced to be that of *initStateFunc*.

*vcella* supports unidimensional cellular automata, where the state of each cell is stored in *ioutFunc*. So *ioutFunc* is a vector containing current state of each cell. This variant vector can be used together with any other vector-based opcode, such as *adsynt*, *vmap*, *vpowv* etc.

*initStateFunc* is an input vector containing the inital value of the row of cells, while *iRuleFunc* is an input vector containing the rules in the form of a lookup table. Notice that *initStateFunc* and *iRuleFunc* can be updated during the performance by means of other vector-based opcodes (for example *vcopy*) in order to force to change rules and status at performance time.

A new generation of cells is evaluated each time *ktrig* contains a non-zero value. Also the status of all cells can be forced to assume the status corresponding to the contents of *initStateFunc* each time *kreinit* contains a non-zero value.

Radius of CA algorithm can be 1 or 2 (optional *iradius* arguement).

## Examples

Here is an example of the *vcella* opcode. It uses the file *vcella.csd* [examples/vcella.csd].

The following example uses *vcella*

### Example 1057. Example of the *vcella* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>

<CsOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in
-odac        -iadc        ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o vcella.wav -W ;; for file output any platform
</CsOptions>

<CsInstruments>
; vcella.csd
; by Anthony Kozar

; This file demonstrates some of the new opcodes available in
; Csound 5 that come from Gabriel Maldonado's CsoundAV.

sr          = 44100
kr          = 4410
ksmps      = 10
nchnls     = 1

; Cellular automata-driven oscillator bank using vcella and adsynt
instr 1
  idur      = p3
  iCArate   = p4                ; number of times per second the CA calculates new values

  ; f-tables for CA parameters
  iCAinit   = p5                ; CA initial states
  iCARule   = p6                ; CA rule values
  ; The rule is used as follows:
  ; the states (values) of each cell are summed with their neighboring cells within
  ; the specied radius (+/- 1 or 2 cells). Each sum is used as an index to read a
  ; value from the rule table which becomes the new state value for its cell.
  ; All new states are calculated first, then the new values are all applied
  ; simultaneously.

  ielements = ftlen(iCAinit)
  inumrules  = ftlen(iCARule)
  iradius    = 1

  ; create some needed tables
  iCAstate   ftgen    0, 0, ielements, -2, 0    ; will hold the current CA states
  ifreqs     ftgen    0, 0, ielements, -2, 0    ; will hold the oscillator frequency for each cell
  iamps      ftgen    0, 0, ielements, -2, 0    ; will hold the amplitude for each cell

  ; calculate cellular automata state
  ktrig      metro    iCArate                ; trigger the CA to update iCArate times per second
             vcella   ktrig, 0, iCAstate, iCAinit, iCARule, ielements, inumrules, iradius

  ; scale CA state for use as amplitudes of the oscillator bank
             vcopy    iamps, iCAstate, ielements
             vmult    iamps, (1/3), ielements    ; divide by 3 since state values are 0-3

             vport    iamps, .01, ielements    ; need to smooth the amplitude changes for adsynt
  ; we could use adsynt2 instead of adsynt, but it does not seem to be working

  ; i-time loop for calculating frequencies

```

```

index      =      0
inew       =      1
iratio     =      1.125           ; just major second (creating a whole tone scale)
loop1:
    tableiw inew, index, ifreqs, 0 ; 0 indicates integer indices
    inew    =      inew * iratio
    index   =      index + 1
    if (index < ielements) igoto loop1

; create sound with additive oscillator bank
ifreqbase = 64
iwavefn   = 1
iphs      = 2           ; random oscillator phases

kenv      linseg  0.0, 0.5, 1.0, idur - 1.0, 1.0, 0.5, 0.0
aosc      adsynt  kenv, ifreqbase, iwavefn, ifreqs, iamps, ielements, iphs

    out     aosc * ampdb(68)

endin

</CsInstruments>

<CsScore>
f1 0 16384 10 1

; This example uses a 4-state cellular automata
; Possible state values are 0, 1, 2, and 3

; CA initial state
; We have 16 cells in our CA, so the initial state table is size 16
f10 0 16 -2 0 1 0 0 1 0 0 2 2 0 0 1 0 0 1 0

; CA rule
; The maximum sum with radius 1 (3 cells) is 9, so we need 10 values in the rule (0-9)
f11 0 16 -2 1 0 3 2 1 0 0 2 1 0

; Here is our one and only note!
i1 0 20 4 10 11

e

</CsScore>
</CsoundSynthesizer>

```

## Credits

Written by: Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

Example by: Anthony Kozar

## VCO

*vco* — Implementation of a band limited, analog modeled oscillator.

## Description

Implementation of a band limited, analog modeled oscillator, based on integration of band limited impulses. *vco* can be used to simulate a variety of analog wave forms.

## Syntax

```
ares vco xamp, xcps, iwave, kpw [, ifn] [, imaxd] [, ileak] [, inyx] \  
    [, iphs] [, iskip]
```

## Initialization

*iwave* -- determines the waveform:

- *iwave* = 1 - sawtooth
- *iwave* = 2 - square/PWM
- *iwave* = 3 - triangle/saw/ramp

*ifn* (optional, default = 1) -- should be the table number of a stored sine wave. Must point to a valid table which contains a sine wave. Csound will report an error if this parameter is not set and table 1 doesn't exist.

*imaxd* (optional, default = 1) -- is the maximum delay time. A time of  $1/ifaq$  may be required for the PWM and triangle waveform. To bend the pitch down this value must be as large as  $1/(\text{minimum frequency})$ .

*ileak* (optional, default = 0) -- if *ileak* is between zero and one ( $0 < ileak < 1$ ) then *ileak* is used as the leaky integrator value. Otherwise a leaky integrator value of .999 is used for the saw and square waves and .995 is used for the triangle wave. This can be used to “flatten” the square wave or “straighten” the saw wave at low frequencies by setting *ileak* to .99999 or a similar value. This should give a hollow sounding square wave.

*inyx* (optional, default = .5) -- this is used to determine the number of harmonics in the band limited pulse. All overtones up to  $sr * inyx$  will be used. The default gives  $sr * .5$  ( $sr/2$ ). For  $sr/4$  use *inyx* = .25. This can generate a “fatter” sound in some cases.

*iphs* (optional, default = 0) -- this is a phase value. There is an artifact (bug-like feature) in *vco* which occurs during the first half cycle of the square wave which causes the waveform to be greater in magnitude than all others. The value of *iphs* has an effect on this artifact. In particular setting *iphs* to .5 will cause the first half cycle of the square wave to resemble a small triangle wave. This may be more desirable than the large wave artifact which is the current default.

*iskip* (optional, default = 0) -- if non zero skip the initialisation of the filter. (New in Csound version 4.23f13 and 5.0)

## Performance

*kpw* -- determines either the pulse width (if *iwave* is 2) or the saw/ramp character (if *iwave* is 3) The value of *kpw* should be greater than 0 and less than 1. A value of 0.5 will generate either a square wave (if *iwave* is 2) or a triangle wave (if *iwave* is 3).

*xamp* -- determines the amplitude

*xcps* -- is the frequency of the wave in cycles per second.

## Examples

Here is an example of the vco opcode. It uses the file *vco.csd* [examples/vco.csd].

### Example 1058. Example of the vco opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o vco.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1
instr 1
; Set the amplitude.
kamp = p4

; Set the frequency.
kcps = cpspch(p5)

; Select the wave form.
iwave = p6

; Set the pulse-width/saw-ramp character.
kpw init 0.5

; Use Table #1.
ifn = 1

; Generate the waveform.
asig vco kamp, kcps, iwave, kpw, ifn

; Output and amplification.
out asig
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 65536 10 1

; Define the score.
; p4 = raw amplitude (0-32767)
; p5 = frequency, in pitch-class notation.
```

```
; p6 = the waveform (1=Saw, 2=Square/PWM, 3=Tri/Saw-Ramp-Mod)
i 1 00 02 20000 05.00 1
i 1 02 02 20000 05.00 2
i 1 04 02 20000 05.00 3

i 1 06 02 20000 07.00 1
i 1 08 02 20000 07.00 2
i 1 10 02 20000 07.00 3

i 1 12 02 20000 09.00 1
i 1 14 02 20000 09.00 2
i 1 16 02 20000 09.00 3

i 1 18 02 20000 11.00 1
i 1 20 02 20000 11.00 2
i 1 22 02 20000 11.00 3
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*vco2*

## Credits

Author: Hans Mikelson  
December 1998

New in Csound version 3.50

November 2002. Corrected the documentation for the *kpw* parameter thanks to Luis Jure and Hans Mikelson.

## vco2

vco2 — Implementation of a band-limited oscillator using pre-calculated tables.

### Description

*vco2* is similar to *vco*. But the implementation uses pre-calculated tables of band-limited waveforms (see also *GEN30*) rather than integrating impulses. This opcode can be faster than *vco* (especially if a low control-rate is used) and also allows better sound quality. Additionally, there are more waveforms and oscillator phase can be modulated at k-rate. The disadvantage is increased memory usage. For more details about *vco2* tables, see also *vco2init* and *vco2ft*.

### Syntax

```
ares vco2 kamp, kcps [, imode] [, kpw] [, kphs] [, inyx]
```

### Initialization

*imode* (optional, default=0) -- a sum of values representing the waveform and its control values.

One may use any of the following values for *imode*:

- 16: enable k-rate phase control (if set, *kphs* is a required k-rate parameter that allows phase modulation)
- 1: skip initialization

One may use exactly one of these *imode* values to select the waveform to be generated:

- 14: user defined waveform -1 (requires using the *vco2init* opcode)
- 12: triangle (no ramp, faster)
- 10: square wave (no PWM, faster)
- 8:  $4 * x * (1 - x)$  (i.e. integrated sawtooth)
- 6: pulse (not normalized)
- 4: sawtooth / triangle / ramp
- 2: square / PWM
- 0: sawtooth

The default value for *imode* is zero, which means a sawtooth wave with no k-rate phase control.

*inyx* (optional, default=0.5) -- bandwidth of the generated waveform, as percentage (0 to 1) of the sample rate. The expected range is 0 to 0.5 (i.e. up to  $sr/2$ ), other values are limited to the allowed range.

Setting *inyx* to 0.25 ( $sr/4$ ), or 0.3333 ( $sr/3$ ) can produce a “fatter” sound in some cases, although it is more likely to reduce quality.

### Performance

*ares* -- the output audio signal.

*kamp* -- amplitude scale. In the case of a *imode* waveform value of 6 (a pulse waveform), the actual output level can be a lot higher than this value.

*kcps* -- frequency in Hz (should be in the range  $-sr/2$  to  $sr/2$ ).

*kpw* (optional) -- the pulse width of the square wave (*imode* waveform=2) or the ramp characteristics of the triangle wave (*imode* waveform=4). It is required only by these waveforms and ignored in all other cases. The expected range is 0 to 1, any other value is wrapped to the allowed range.



## Warning

*kpw* must not be an exact integer value (e.g. 0 or 1) if a sawtooth / triangle / ramp (*imode* waveform=4) is generated. In this case, the recommended range is about 0.01 to 0.99. There is no such limitation for a square/PWM waveform.

*kphs* (optional) -- oscillator phase (depending on *imode*, this can be either an optional i-rate parameter that defaults to zero or required k-rate). Similarly to *kpw*, the expected range is 0 to 1.



## Note

When a low control-rate is used, pulse width (*kpw*) and phase (*kphs*) modulation is internally converted to frequency modulation. This allows for faster processing and reduced artifacts. But in the case of very long notes and continuous fast changes in *kpw* or *kphs*, the phase may drift away from the requested value. In most cases, the phase error is at most 0.037 per hour (assuming a sample rate of 44100 Hz).

This is a problem mainly in the case of pulse width (*kpw*), where it may result in various artifacts. While future releases of *vco2* may fix such errors, the following work-arounds may also be of some help:

- Use *kpw* values only in the range 0.05 to 0.95. (There are more artifacts around integer values)
- Try to avoid modulating *kpw* by asymmetrical waveforms like a sawtooth wave. Relatively slow ( $\leq 20$  Hz) symmetrical modulation (e.g. sine or triangle), random splines (also slow), or a fixed pulse width is a lot less likely to cause synchronization problems.
- In some cases, adding random jitter (for example: random splines with an amplitude of about 0.01) to *kpw* may also fix the problem.

## Examples

Here is an example of the *vco2* opcode. It uses the file *vco2.csd* [examples/vco2.csd].

### Example 1059. Example of the *vco2* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc    -d      ;;RT audio I/O
```

```

; For Non-realtime ouput leave only the line below:
; -o vco2.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr      = 44100
ksmps  = 10
nchnls = 1

; user defined waveform -1: trapezoid wave with default parameters (can be
; accessed at ftables starting from 10000)
itmp   ftgen 1, 0, 16384, 7, 0, 2048, 1, 4096, 1, 4096, -1, 4096, -1, 2048, 0
ift    vco2init -1, 10000, 0, 0, 0, 1
; user defined waveform -2: fixed table size (4096), number of partials
; multiplier is 1.02 (~238 tables)
itmp   ftgen 2, 0, 16384, 7, 1, 4095, 1, 1, -1, 4095, -1, 1, 0, 8192, 0
ift    vco2init -2, ift, 1.02, 4096, 4096, 2

instr 1
kcps   expon p4, p3, p5           ; instr 1: basic vco2 example
a1     vco2 12000, kcps           ; (sawtooth wave with default
out a1                               ; parameters)
endin

instr 2
kcps   expon p4, p3, p5           ; instr 2:
kpw    linseg 0.1, p3/2, 0.9, p3/2, 0.1 ; PWM example
a1     vco2 10000, kcps, 2, kpw
out a1
endin

instr 3
kcps   expon p4, p3, p5           ; instr 3: vco2 with user
a1     vco2 14000, kcps, 14       ; defined waveform (-1)
aenv   linseg 1, p3 - 0.1, 1, 0.1, 0 ; de-click envelope
out a1 * aenv
endin

instr 4
kcps   expon p4, p3, p5           ; instr 4: vco2ft example,
kfn    vco2ft kcps, -2, 0.25     ; with user defined waveform
a1     oscilikt 12000, kcps, kfn ; (-2), and sr/4 bandwidth
out a1
endin

</CsInstruments>
<CsScore>

i 1 0 3 20 2000
i 2 4 2 200 400
i 3 7 3 400 20
i 4 11 2 100 200

f 0 14

e

</CsScore>
</CsoundSynthesizer>

```

## See Also

*vco*, *vco2ft*, *vco2ift*, and *vco2init*.

## Credits

Author: Istvan Varga

New in version 4.22

## vco2ft

vco2ft — Returns a table number at k-time for a given oscillator frequency and wavform.

### Description

*vco2ft* returns the function table number to be used for generating the specified waveform at a given frequency. This function table number can be used by any Csound opcode that generates a signal by reading function tables (like *oscilikt*). The tables must be calculated by *vco2init* before *vco2ft* is called and shared as Csound ftables (*ibasfn*).

### Syntax

```
kfn vco2ft kcps, iwave [, inyx]
```

### Initialization

*iwave* -- the waveform for which table number is to be selected. Allowed values are:

- 0: sawtooth
- 1:  $4 * x * (1 - x)$  (integrated sawtooth)
- 2: pulse (not normalized)
- 3: square wave
- 4: triangle

Additionally, negative *iwave* values select user defined waveforms (see also *vco2init*).

*inyx* (optional, default=0.5) -- bandwidth of the generated waveform, as percentage (0 to 1) of the sample rate. The expected range is 0 to 0.5 (i.e. up to  $sr/2$ ), other values are limited to the allowed range.

Setting *inyx* to 0.25 ( $sr/4$ ), or 0.3333 ( $sr/3$ ) can produce a “fatter” sound in some cases, although it is more likely to reduce quality.

### Performance

*kfn* -- the ftable number, returned at k-rate.

*kcps* -- frequency in Hz, returned at k-rate. Zero and negative values are allowed. However, if the absolute value exceeds  $sr/2$  (or  $sr * inyx$ ), the selected table will contain silence.

### Examples

See the example for the *vco2* opcode.

### See Also

*vco2ift*, *vco2init*, and *vco2*.

## Credits

Author: Istvan Varga

New in version 4.22

# vco2ift

vco2ift — Returns a table number at i-time for a given oscillator frequency and wavform.

## Description

*vco2ift* is the same as *vco2ft*, but works at i-time. It is suitable for use with opcodes that expect an i-rate table number (for example, *oscili*).

## Syntax

```
ifn vco2ift icps, iwave [, inyx]
```

## Initialization

*ifn* -- the ftable number.

*icps* -- frequency in Hz. Zero and negative values are allowed. However, if the absolute value exceeds  $sr/2$  (or  $sr * inyx$ ), the selected table will contain silence.

*iwave* -- the waveform for which table number is to be selected. Allowed values are:

- 0: sawtooth
- 1:  $4 * x * (1 - x)$  (integrated sawtooth)
- 2: pulse (not normalized)
- 3: square wave
- 4: triangle

Additionally, negative *iwave* values select user defined waveforms (see also *vco2init*).

*inyx* (optional, default=0.5) -- bandwidth of the generated waveform, as percentage (0 to 1) of the sample rate. The expected range is 0 to 0.5 (i.e. up to  $sr/2$ ), other values are limited to the allowed range.

Setting *inyx* to 0.25 ( $sr/4$ ), or 0.3333 ( $sr/3$ ) can produce a “fatter” sound in some cases, although it is more likely to reduce quality.

## Examples

Here is an example of the vco2ift opcode. It uses the file *vco2ift.csd* [examples/vco2ift.csd].

### Example 1060. Example of the vco2ift opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform
```

```
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o vco2ift.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

; user defined waveform -2: fixed table size (64), number of partials
; multiplier is 1.4
itmp   ftgen 2, 0, 64, 5, 1, 2, 120, 60, 1, 1, 0.001, 1
ift    vco2init -2, 2, 1.4, 4096, 4096, 2

instr 1

icps = p4
ifn   vco2ift icps, -2, 0.5 ;with user defined waveform
print ifn
asig  oscili 1, 220, ifn ; (-2), and sr/2 bandwidth
      outs asig, asig

endin
</CsInstruments>
<CsScore>

i 1 0 2 20
i 1 3 2 2000
i 1 6 2 20000

e
</CsScore>
</CsSoundSynthesizer>
```

Its output should include a line like these:

```
instr 1: ifn = 22.000
instr 1: ifn = 8.000
instr 1: ifn = 3.000
```

See the example for the `vco2` opcode too.

## See Also

`vco2ft`, `vco2init`, and `vco2`.

## Credits

Author: Istvan Varga

New in version 4.22

# vco2init

vco2init — Calculates tables for use by vco2 opcode.

## Description

*vco2init* calculates tables for use by *vco2* opcode. Optionally, it is also possible to access these tables as standard Csound function tables. In this case, *vco2ft* can be used to find the correct table number for a given oscillator frequency.

In most cases, this opcode is called from the orchestra header. Using *vco2init* in instruments is possible but not recommended. This is because replacing tables during performance can result in a Csound crash if other opcodes are accessing the tables at the same time.

Note that *vco2init* is not required for *vco2* to work (tables are automatically allocated by the first *vco2* call, if not done yet), however it can be useful in some cases:

- Pre-calculate tables at orchestra load time. This is useful to avoid generating the tables during performance, which could interrupt real-time processing.
- Share the tables as Csound ftables. By default, the tables can be accessed only by *vco2*.
- Change the default parameters of tables (e.g. size) or use an user-defined waveform specified in a function table.

## Syntax

```
ifn vco2init iwave [, ibasfn] [, ipmul] [, iminsiz] [, imaxsiz] [, isrcft]
```

## Initialization

*ifn* -- the first free ftable number after the allocated tables. If *ibasfn* was not specified, -1 is returned.

*iwave* -- sum of the following values selecting which waveforms are to be calculated:

- 16: triangle
- 8: square wave
- 4: pulse (not normalized)
- 2:  $4 * x * (1 - x)$  (integrated sawtooth)
- 1: sawtooth

Alternatively, *iwave* can be set to a negative integer that selects an user-defined waveform. This also requires the *isrcft* parameter to be specified. *vco2* can access waveform number -1. However, other user-defined waveforms are usable only with *vco2ft* or *vco2ift*.

*ibasfn* (optional, default=-1) -- ftable number from which the table set(s) can be accessed by opcodes other than *vco2*. This is required by user defined waveforms, with the exception of -1. If this value is less than 1, it is not possible to access the tables calculated by *vco2init* as Csound function tables.

*ipmul* (optional, default=1.05) -- multiplier value for number of harmonic partials. If one table has  $n$  partials, the next one will have  $n * ipmul$  (at least  $n + 1$ ). The allowed range for *ipmul* is 1.01 to 2. Zero or negative values select the default (1.05).

*iminsiz* (optional, default=-1) -- minimum table size.

*imaxsiz* (optional, default=-1) -- maximum table size.

The actual table size is calculated by multiplying the square root of the number of harmonic partials by *iminsiz*, rounding up the result to the next power of two, and limiting this not to be greater than *imaxsiz*.

Both parameters, *iminsiz* and *imaxsiz*, must be power of two, and in the allowed range. The allowed range is 16 to 262144 for *iminsiz* to up to 16777216 for *imaxsiz*. Zero or negative values select the default settings:

- The minimum size is 128 for all waveforms except pulse (*iwave*=4). Its minimum size is 256.
- The default maximum size is usually the minimum size multiplied by 64, but not more than 16384 if possible. It is always at least the minimum size.

*isrcft* (optional, default=-1) -- source ftable number for user-defined waveforms (if *iwave* < 0). *isrcft* should point to a function table containing the waveform to be used for generating the table array. The table size is recommended to be at least *imaxsiz* points. If *iwave* is not negative (built-in waveforms are used), *isrcft* is ignored.



## Warning

The number and size of tables is not fixed. Orchestras should not depend on these parameters, as they are subject to changes between releases.

If the selected table set already exists, it is replaced. If any opcode is accessing the tables at the same time, it is very likely that a crash will occur. This is why it is recommended to use *vco2init* only in the orchestra header.

These tables should not be replaced/overwritten by GEN routines or the *fgen* opcode. Otherwise, unpredictable behavior or a Csound crash may occur if *vco2* is used. The first free ftable after the table array(s) is returned in *ifn*.

## Examples

Here is an example of the *vco2init* opcode. It uses the file *vco2init.csd* [examples/vco2init.csd].

### Example 1061. Example of the *vco2init* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac ;;realtime audio out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o vco2init.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr=44100
```

```
ksmps=1
nchnls=2

; create waveform with discontinuities, so it has a lot of high freq content
gitable ftgen 0, 0, 2^16+1, 7, -1, 2^14, 1, 0, -1, 2^14, 1, 0, -1, 2^15, 1
; make bandlimited tables of the waveform
gi_nextfree vco2init -gitable, gitable+1, 1.05, 128, 2^16, gitable
gitable_bl = -gitable

instr 1

kfreq expon 14000, p3, 500
kfn vco2ft kfreq, gitable_bl
asig oscilikt 5000, kfreq, kfn
printk 0.1, kfn

; remove semicolon on next line to hear original waveform, demonstrating the aliasing
;asig oscili 5000, kfreq, gitable
outs asig, asig

endin
</CsInstruments>
<CsScore>
i1 0 5
e
</CsScore>
</CsoundSynthesizer>
```

Its output should include a line like these:

```
i 1 time 0.00002: 103.00000
i 1 time 0.10000: 103.00000
i 1 time 0.20000: 103.00000
i 1 time 0.30002: 103.00000
i 1 time 0.40000: 104.00000
i 1 time 0.50000: 104.00000
.....
.....
i 1 time 4.80002: 135.00000
i 1 time 4.90000: 136.00000
i 1 time 5.00000: 138.00000
```

See the example for the `vco2` opcode too.

## See Also

`vco2ft`, `vco2ift`, and `vco2`.

## Credits

Author: Istvan Varga

New in version 4.22

# vcomb

vcomb — Variably reverberates an input signal with a “colored” frequency response.

## Description

Variably reverberates an input signal with a “colored” frequency response.

## Syntax

```
ares vcomb asig, krvt, xlpt, imaxlpt [, iskip] [, insmps]
```

## Initialization

*imaxlpt* -- maximum loop time for *klpt*

*iskip* (optional, default=0) -- initial disposition of delay-loop data space (cf. *reson*). The default value is 0.

*insmps* (optional, default=0) -- delay amount, as a number of samples.

## Performance

*krvt* -- the reverberation time (defined as the time in seconds for a signal to decay to 1/1000, or 60dB down from its original amplitude).

*xlpt* -- variable loop time in seconds, same as *ilpt* in *comb*. Loop time can be as large as *imaxlpt*.

This filter reiterates input with an echo density determined by loop time *xlpt*. The attenuation rate is independent and is determined by *krvt*, the reverberation time (defined as the time in seconds for a signal to decay to 1/1000, or 60dB down from its original amplitude). Output will appear only after *ilpt* seconds.

## Examples

Here is an example of the vcomb opcode. It uses the file *vcomb.csd* [examples/vcomb.csd].

### Example 1062. Example of the vcomb opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc          -M0 ;;;RT audio I/O with MIDI in
</CsOptions>
<CsInstruments>

; Example by Jonathan Murphy and Charles Gran 2007
sr      = 44100
ksmps  = 10
nchnls = 2
```

```

; new, and important. Make sure that midi note events are only
; received by instruments that actually need them.

; turn default midi routing off
massign 0, 0
; route note events on channel 1 to instr 1
massign 1, 1

; Define your midi controllers
#define C1 #21#
#define C2 #22#
#define C3 #23#

; Initialize MIDI controllers
initc7 1, $C1, 0.5 ;delay send
initc7 1, $C2, 0.5 ;delay: time to zero
initc7 1, $C3, 0.5 ;delay: rate

gaosc   init      0

; Define an opcode to "smooth" the MIDI controller signal
opcode smooth, k, k
kin     xin
kport  linseg    0, 0.0001, 0.01, 1, 0.01
kin     portk    kin, kport
xout    xout     kin
endop

instr 1
; Generate a sine wave at the frequency of the MIDI note that triggered the instrument
ifqc   cpsmidi
iamp   ampmidi  10000
aenv   linenr   iamp, .01, .1, .01 ;envelope
a1     oscil    aenv, ifqc, 1
; All sound goes to the global variable gaosc
gaosc  = gaosc + a1
endin

instr 198 ; ECHO
kcbsnd ctrl7    1, $C1, 0, 1 ;delay send
ktime  ctrl7    1, $C2, 0.01, 6 ;time loop fades out
kloop  ctrl7    1, $C3, 0.01, 1 ;loop speed
; Receive MIDI controller values and then smooth them
kcbsnd smooth   kcbsnd
ktime  smooth   ktime
kloop  smooth   kloop
imaxlpt = 1 ;max loop time
; Create a variable reverberation (delay) of the gaosc signal
acomb  vcomb    gaosc, ktime, kloop, imaxlpt, 1
aout   = (acomb * kcbsnd) + gaosc * (1 - kcbsnd)
outs   aout, aout
gaosc  = 0
endin

</CsInstruments>

<CsScore>
f1 0 16384 10 1
i198 0 10000
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*alpass, comb, reverb, valpass*

## Credits

Author: William "Pete" Moss  
University of Texas at Austin  
Austin, Texas USA  
January 2002

# vcopy

vcopy — Copies between two vectorial control signals

## Description

Copies between two vectorial control signals

## Syntax

```
vcopy ifn1, ifn2, kelements [, kdstoffset] [, ksrcoffset] [, kverbose]
```

## Initialization

*ifn1* - number of the table where the vectorial signal will be copied (destination)

*ifn2* - number of the table hosting the vectorial signal to be copied (source)

## Performance

*kelements* - number of elements of the vector

*kdstoffset* - index offset for the destination (*ifn1*) table (Default=0)

*ksrcoffset* - index offset for the source (*ifn2*) table (Default=0)

*kverbose* - Selects whether or not warnings are printed (Default=0)

*vcopy* copies *kelements* elements from *ifn2* (starting from position *ksrcoffset*) to *ifn1* (starting from position *kdstoffset*). Useful to keep old vector values, by storing them in another table.

Negative values for *kdstoffset* and *ksrcoffset* are acceptable. If *kdstoffset* is negative, the out of range section of the vector will be discarded. If *kdstoffset* is negative, the out of range elements will be assumed to be 1 (i.e. the destination elements will not be changed). If elements for the destination vector are beyond the size of the table (including guard point), these elements are discarded (i.e. elements do not wrap around the tables). If elements for the source vector are beyond the table length, these elements are taken as 1 (i.e. the destination vector will not be changed for these elements).

If the optional *kverbose* argument is different to 0, the opcode will print warning messages every k-pass if table lengths are exceeded.



### Warning

Using the same table as source and destination table in versions earlier than 5.04, might produce unexpected behavior, so use with care.

This opcode works at k-rate (this means that every k-pass the vectors are copied). There's an i-rate version of this opcode called *vcopy\_i*.



### Note

Please note that the *elements* argument has changed in version 5.03 from i-rate to k-rate. This will change the opcode's behavior in the unusual cases where the i-rate variable *ielements* is changed inside the instrument, for example in:

```

instr 1
ielements = 10
vadd 1, 1, ielements
ielements = 20
vadd 2, 1, ielements
turnoff
endin

```

All these operators (*vaddv*, *vsubv*, *vmultv*, *vdivv*, *vpowv*, *vexp*, *vcopy* and *vmap*) are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2* etc.

## Examples

Here is an example of the *vcopy* opcode. It uses the file *vcopy.csd* [examples/vcopy.csd].

### Example 1063. Example of the *vcopy* opcode.

```

<CsoundSynthesizer>
<CsOptions>; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o vcopy.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr=44100
kr=4410
ksmps=10
nchnls=2

instr 1 ;table playback
ar lposcil 1, 1, 0, 262144, 1
outs ar,ar
endin

instr 2
vcopy 2, 1, 20000 ;copy vector from sample to empty table
vmult 5, 20000, 262144 ;scale noise to make it audible
vcopy 1, 5, 20000 ;put noise into sample
turnoff
endin

instr 3
vcopy 1, 2, 20000 ;put original information back in
turnoff
endin

</CsInstruments>
<CsScore>
f1 0 262144 -1 "beats.wav" 0 4 0
f2 0 262144 2 0

f5 0 262144 21 3 30000

i1 0 4
i2 3 1

s
i1 0 4

```

```
i3 3 1  
s  
  
i1 0 4  
  
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado. Optional arguments added by Andres Cabrera and Istvan Varga.

New in Csound 5 (Previously available only on CsoundAV)

# vcopy\_i

`vcopy_i` — Copies a vector from one table to another.

## Description

Copies a vector from one table to another.

## Syntax

```
vcopy_i ifn1, ifn2, ielements [,idstoffset, isrcoffset]
```

## Initialization

*ifn1* - number of the table where the vectorial signal will be copied

*ifn2* - number of the table hosting the vectorial signal to be copied

*ielements* - number of elements of the vector

*idstoffset* - index offset for destination table

*isrcoffset* - index offset for source table

## Performance

`vcopy_i` copies *ielements* elements from *ifn2* (starting from position *isrcoffset*) to *ifn1* (starting from position *idstoffset*). Useful to keep old vector values, by storing them in another table. This opcode is exactly the same as `vcopy` but performs all the copying on the initialization pass only.

Negative values for *idstoffset* and *isrcoffset* are acceptable. If *idstoffset* is negative, the out of range section of the vector will be discarded. If *isrcoffset* is negative, the out of range elements will be assumed to be 0 (i.e. the destination elements will be set to 0). If elements for the destination vector are beyond the size of the table (including guard point), these elements are discarded (i.e. elements do not wrap around the tables). If elements for the source vector are beyond the table length, these elements are taken as 0 (i.e. the destination vector elements will be 0).



### Warning

Using the same table as source and destination table in versions earlier than 5.04, might produce unexpected behavior, so use with care.

All these operators (`vaddv`, `vsubv`, `vmultv`, `vdivv`, `vpowv`, `vexp`, `vcopy` and `vmap`) are designed to be used together with other opcodes that operate with vectorial signals such as `vcella`, `adsynt`, `adsynt2` etc.

## Examples

See `vcopy` for an example.

## Credits

Written by Gabriel Maldonado. Optional arguments added by Andres Cabrera and Istvan Varga.

New in Csound 5 (Previously available only on CsoundAV)

# vdelay

vdelay — An interpolating variable time delay.

## Description

This is an interpolating variable time delay, it is not very different from the existing implementation (*deltapi*), it is only easier to use.

## Syntax

```
ares vdelay asig, adel, imaxdel [, iskip]
```

## Initialization

*imaxdel* -- Maximum value of delay in milliseconds. If *adel* gains a value greater than *imaxdel* it is folded around *imaxdel*. This should not happen.

*iskip* -- Skip initialization if present and nonzero

## Performance

With this unit generator it is possible to do Doppler effects or chorusing and flanging.

*asig* -- Input signal.

*adel* -- Current value of delay in milliseconds. Note that linear functions have no pitch change effects. Fast changing values of *adel* will cause discontinuities in the waveform resulting noise.

## Examples

Here is an example of the vdelay opcode. It uses the file *vdelay.csd* [examples/vdelay.csd].

### Example 1064. Example of the vdelay opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o vdelay.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr      = 44100
ksmps   = 32
nchnls  = 2
0dbfs   = 1
```

```
instr 1

ims = 100      ;maximum delay time in msec
aout poscil .8, 220, 1 ;make a signal
a2 poscil3 ims/2, 1/p3, 1 ;make an LFO
a2 = a2 + ims/2 ;offset the LFO so that it is positive
asig vdelay aout, a2, ims ;use the LFO to control delay time
outs asig, asig

endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1 ;sine wave

i 1 0 5

e
</CsScore>
</CsoundSynthesizer>
```

Two important points here. First, the delay time must be always positive. And second, even though the delay time can be controlled in k-rate, it is not advised to do so, since sudden time changes will create clicks.

## See Also

*vdelay3*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

# vdelay3

vdelay3 — A variable time delay with cubic interpolation.

## Description

*vdelay3* is experimental. It is the same as *vdelay* except that it uses cubic interpolation. (New in Version 3.50.)

## Syntax

```
ares vdelay3 asig, adel, imaxdel [, iskip]
```

## Initialization

*imaxdel* -- Maximum value of delay in milliseconds. If *adel* gains a value greater than *imaxdel* it is folded around *imaxdel*. This should not happen.

*iskip* (optional) -- Skip initialization if present and non-zero.

## Performance

With this unit generator it is possible to do Doppler effects or chorusing and flanging.

*asig* -- Input signal.

*adel* -- Current value of delay in milliseconds. Note that linear functions have no pitch change effects. Fast changing values of *adel* will cause discontinuities in the waveform resulting noise.

## Examples

Here is an example of the *vdelay3* opcode. It uses the file *vdelay3.csd* [examples/vdelay3.csd].

### Example 1065. Example of the vdelay3 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o vdelay3.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
```

```
instr 1

ims = 100 ;maximum delay time in msec
aout poscil .8, 220, 1 ;make a signal
a2 poscil ims/2, 1/p3, 1 ;make an LFO
a2 = a2 + ims/2 ;offset the LFO so that it is positive
asig vdelay3 aout, a2, ims ;use the LFO to control delay time
outs asig, asig

endin

</CsInstruments>
<CsScore>
f1 0 8192 10 1

i 1 0 5

e

</CsScore>
</CsoundSynthesizer>
```

Two important points here. First, the delay time must be always positive. And second, even though the delay time can be controlled in k-rate, it is not advised to do so, since sudden time changes will create clicks.

## See Also

*vdelay*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

# vdelayx

vdelayx — A variable delay opcode with high quality interpolation.

## Description

A variable delay opcode with high quality interpolation.

## Syntax

```
aout vdelayx ain, adl, imd, iws [, ist]
```

## Initialization

*imd* -- max. delay time (seconds)

*iws* -- interpolation window size (see below)

*ist* (optional) -- skip initialization if not zero

## Performance

*aout* -- output audio signal

*ain* -- input audio signal

*adl* -- delay time in seconds

This opcode uses high quality (and slow) interpolation, that is much more accurate than the currently available linear and cubic interpolation. The *iws* parameter sets the number of input samples used for calculating one output sample (allowed values are any integer multiply of 4 in the range 4 - 1024); higher values mean better quality and slower speed.



### Notes

- Delay time is measured in seconds (unlike in `vdelay` and `vdelay3`), and must be a-rate.
- The minimum allowed delay is  $iws/2$  samples.
- Using the same variables as input and output is allowed in these opcodes.
- In `vdelayxw*`, changing the delay time has some effects on output volume:  
$$a = 1 / (1 + dt)$$
where  $a$  is the output gain, and  $dt$  is the change of delay time per seconds.
- These opcodes are best used in the double-precision version of Csound.

## Examples

Here is an example of the use of the `vdelayx` opcode. It uses the file `vdelayx.csd` [examples/vdelayx.csd].

### Example 1066. Example of the *vdelayx* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o vdelayx.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ims = .5      ;maximum delay time in seconds
iws = 1024    ;window size
adl = .5      ;delay time
asig diskin2 "fox.wav", 1, 0, 1 ;loop fox.wav
a2 poscil .2, .2, 1 ;make an LFO
adl = a2 + ims/2 ;offset the LFO so that it is positive
aout vdelayx asig, adl, ims, iws ;use the LFO to control delay time
outs aout, aout

endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1

i 1 0 10

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*vdelayxq*, *vdelayxs*, *vdelayxw*, *vdelayxwq*, *vdelayxws*

# vdelayxq

vdelayxq — A 4-channel variable delay opcode with high quality interpolation.

## Description

A 4-channel variable delay opcode with high quality interpolation.

## Syntax

```
aout1, aout2, aout3, aout4 vdelayxq ain1, ain2, ain3, ain4, adl, imd, iws [, ist]
```

## Initialization

*imd* -- max. delay time (seconds)

*iws* -- interpolation window size (see below)

*ist* (optional) -- skip initialization if not zero

## Performance

*aout1*, *aout2*, *aout3*, *aout4* -- output audio signals.

*ain1*, *ain2*, *ain3*, *ain4* -- input audio signals.

*adl* -- delay time in seconds

This opcode uses high quality (and slow) interpolation, that is much more accurate than the currently available linear and cubic interpolation. The *iws* parameter sets the number of input samples used for calculating one output sample (allowed values are any integer multiply of 4 in the range 4 - 1024); higher values mean better quality and slower speed.

The multichannel opcodes (eg. *vdelayxq*) allow delaying 2 or 4 variables at once (stereo or quad signals); this is much more efficient than using separate opcodes for each channel.



### Notes

- Delay time is measured in seconds (unlike in *vdelay* and *vdelay3*), and must be a-rate.
- The minimum allowed delay is  $iws/2$  samples.
- Using the same variables as input and output is allowed in these opcodes.
- In *vdelayxw\**, changing the delay time has some effects on output volume:  
$$a = 1 / (1 + dt)$$
where *a* is the output gain, and *dt* is the change of delay time per seconds.
- These opcodes are best used in the double-precision version of Csound.

## Examples

Here is an example of the `vdelayxq` opcode. It uses the file `vdelayxq.csd` [examples/vdelayxq.csd].

### Example 1067. Example of the `vdelayxq` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac ;;realtime audio out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o vdelayxq.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 4
0dbfs = 1

instr 1

ims = .5 ;maximum delay time in seconds
iws = 1024 ;window size
adl = .5
aout1 diskin2 "beats.wav", 1, 0, 1 ;loop beats.wav
aout2 diskin2 "fox.wav", 1, 0, 1 ;loop fox.wav
aout3 diskin2 "Church.wav", 1, 0, 1 ;loop Church.wav
aout4 diskin2 "flute.aiff", 1, 0, 1 ;loop flute.aiff
a2 poscil .1, .5, 1 ;make an LFO, 1 cycle per 2 seconds
adl = a2 + ims/2 ;offset the LFO so that it is positive
aout1, aout2, aout3, aout4 vdelayxq aout1, aout2, aout3, aout4, adl, ims, iws; Use the LFO to control
outq aout1, aout2, aout3, aout4

endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1

i 1 0 10

e
</CsScore>
</CsoundSynthesizer>
```

Two important points here. First, the delay time must be always positive. And second, even though the delay time can be controlled in k-rate, it is not advised to do so, since sudden time changes will create clicks.

## See Also

`vdelayx`, `vdelayxs`, `vdelayxw`, `vdelayxwq`, `vdelayxws`

# vdelayxs

`vdelayxs` — A stereo variable delay opcode with high quality interpolation.

## Description

A stereo variable delay opcode with high quality interpolation.

## Syntax

```
aout1, aout2 vdelayxs ain1, ain2, adl, imd, iws [, ist]
```

## Initialization

*imd* -- max. delay time (seconds)

*iws* -- interpolation window size (see below)

*ist* (optional) -- skip initialization if not zero

## Performance

*aout1*, *aout2* -- output audio signals

*ain1*, *ain2* -- input audio signals

*adl* -- delay time in seconds

This opcode uses high quality (and slow) interpolation, that is much more accurate than the currently available linear and cubic interpolation. The *iws* parameter sets the number of input samples used for calculating one output sample (allowed values are any integer multiply of 4 in the range 4 - 1024); higher values mean better quality and slower speed.

The multichannel opcodes (eg. *vdelayxq*) allow delaying 2 or 4 variables at once (stereo or quad signals); this is much more efficient than using separate opcodes for each channel.



### Notes

- Delay time is measured in seconds (unlike in *vdelay* and *vdelay3*), and must be a-rate.
- The minimum allowed delay is  $iws/2$  samples.
- Using the same variables as input and output is allowed in these opcodes.
- In *vdelayxw\**, changing the delay time has some effects on output volume:  
$$a = 1 / (1 + dt)$$
where *a* is the output gain, and *dt* is the change of delay time per seconds.
- These opcodes are best used in the double-precision version of Csound.

## Examples

Here is an example of the use of the `vdelayxs` opcode. It uses the file `vdelayxs.csd` [examples/vdelayxs.csd].

### Example 1068. Example of the `vdelayxs` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac ;;realtime audio out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o vdelayxs.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ims = .5 ;maximum delay time in seconds
iws = 1024 ;window size
adl = .5 ;delay time
asig1, asig2 diskin2 "kickroll.wav", 1, 0, 1 ;loop stereo file kickroll.wav
a2 poscil .25, .1, 1 ;make an LFO, 1 cycle per 2 seconds
adl = a2 + ims/2 ;offset the LFO so that it is positive
aoutL, aoutR vdelayxs asig1, asig2, adl, ims, iws ;use the LFO to control delay time
outs aoutL, aoutR

endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1

i 1 0 10

e
</CsScore>
</CsoundSynthesizer>

```

## See Also

`vdelayx`, `vdelayxq`, `vdelayxw`, `vdelayxwq`, `vdelayxws`

# vdelayxw

vdelayxw — Variable delay opcodes with high quality interpolation.

## Description

Variable delay opcodes with high quality interpolation.

## Syntax

```
aout vdelayxw ain, adl, imd, iws [, ist]
```

## Initialization

*imd* -- max. delay time (seconds)

*iws* -- interpolation window size (see below)

*ist* (optional) -- skip initialization if not zero

## Performance

*aout* -- output audio signal

*ain* -- input audio signal

*adl* -- delay time in seconds

These opcodes use high quality (and slow) interpolation, that is much more accurate than the currently available linear and cubic interpolation. The *iws* parameter sets the number of input samples used for calculating one output sample (allowed values are any integer multiply of 4 in the range 4 - 1024); higher values mean better quality and slower speed.

The *vdelayxw* opcodes change the position of the write tap in the delay line (unlike all other delay ugens that move the read tap), and are most useful for implementing Doppler effects where the position of the listener is fixed, and the sound source is moving.



### Notes

- Delay time is measured in seconds (unlike in *vdelay* and *vdelay3*), and must be a-rate.
- The minimum allowed delay is  $iws/2$  samples.
- Using the same variables as input and output is allowed in these opcodes.
- In *vdelayxw\**, changing the delay time has some effects on output volume:  
$$a = 1 / (1 + dt)$$
where  $a$  is the output gain, and  $dt$  is the change of delay time per seconds.
- These opcodes are best used in the double-precision version of Csound.

## Examples

Here is an example of the use of the `vdelayxw` opcode. It uses the file `vdelayxw.csd` [examples/vdelayxw.csd].

### Example 1069. Example of the `vdelayxw` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac ;;realtime audio out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o vdelayxw.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ims = .5 ;maximum delay time in seconds
iws = 1024 ;best quality
adl = .5 ;delay time
asig diskin2 "flute.aiff", .5, 0, 1 ;loop flute.aiff at half speed
a2 poscil3 .2, .1, 1 ;make an LFO, 1 cycle per 2 seconds
adl = a2 + ims/2 ;offset the LFO so that it is positive
aout vdelayxw asig, adl, ims, iws ;use the LFO to control delay time
outs aout, aout

endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1

i 1 0 10

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

`vdelayx`, `vdelayxq`, `vdelayxs`, `vdelayxwq`, `vdelayxws`

# vdelayxwq

vdelayxwq — Variable delay opcodes with high quality interpolation.

## Description

Variable delay opcodes with high quality interpolation.

## Syntax

```
aout1, aout2, aout3, aout4 vdelayxwq ain1, ain2, ain3, ain4, adl, \  
imd, iws [, ist]
```

## Initialization

*imd* -- max. delay time (seconds)

*iws* -- interpolation window size (see below)

*ist* (optional) -- skip initialization if not zero

## Performance

*ain1, ain2, ain3, ain4* -- input audio signals

*aout1, aout2, aout3, aout4* -- output audio signals

*adl* -- delay time in seconds

These opcodes use high quality (and slow) interpolation, that is much more accurate than the currently available linear and cubic interpolation. The *iws* parameter sets the number of input samples used for calculating one output sample (allowed values are any integer multiply of 4 in the range 4 - 1024); higher values mean better quality and slower speed.

The *vdelayxw* opcodes change the position of the write tap in the delay line (unlike all other delay ugens that move the read tap), and are most useful for implementing Doppler effects where the position of the listener is fixed, and the sound source is moving.

The multichannel opcodes (eg. *vdelayxq*) allow delaying 2 or 4 variables at once (stereo or quad signals); this is much more efficient than using separate opcodes for each channel.



### Notes

- Delay time is measured in seconds (unlike in *vdelay* and *vdelay3*), and must be a-rate.
- The minimum allowed delay is  $iws/2$  samples.
- Using the same variables as input and output is allowed in these opcodes.
- In *vdelayxw\**, changing the delay time has some effects on output volume:

$$a = 1 / (1 + dt)$$

where  $a$  is the output gain, and  $dt$  is the change of delay time per seconds.

- These opcodes are best used in the double-precision version of Csound.

## Examples

Here is an example of the use of the `vdelayxwq` opcode. It uses the file `vdelayxwq.csd` [examples/vdelayxwq.csd].

### Example 1070. Example of the `vdelayxwq` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o vdelayxwq.wav -W   ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 4
0dbfs = 1

instr 1

ims = .5   ;maximum delay time in seconds
iws = 1024 ;best quality
adl = .5   ;delay time
aout1 diskin2 "beats.wav", 1, 0, 1 ;loop beats.wav
aout2 diskin2 "fox.wav", 1, 0, 1 ;loop fox.wav
aout3 diskin2 "Church.wav", 1, 0, 1 ;loop Church.wav
aout4 diskin2 "flute.aiff", 1, 0, 1 ;loop flute.aiff
a2   poscil3 .2, .1, 1 ;make an LFO, 1 cycle per 2 seconds
adl = a2 + ims/2 ;offset the LFO so that it is positive
aout1, aout2, aout3, aout4 vdelayxwq aout1, aout2, aout3, aout4, adl, ims, iws ;use the LFO to control
      outq aout1, aout2, aout3, aout4

endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1

i 1 0 10

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

`vdelayx`, `vdelayxq`, `vdelayxs`, `vdelayxw`, `vdelayxws`

# vdelayxws

vdelayxws — Variable delay opcodes with high quality interpolation.

## Description

Variable delay opcodes with high quality interpolation.

## Syntax

```
aout1, aout2 vdelayxws ain1, ain2, adl, imd, iws [, ist]
```

## Initialization

*imd* -- max. delay time (seconds)

*iws* -- interpolation window size (see below)

*ist* (optional) -- skip initialization if not zero

## Performance

*ain1, ain2* -- input audio signals

*aout1, aout2* -- output audio signals

*adl* -- delay time in seconds

These opcodes use high quality (and slow) interpolation, that is much more accurate than the currently available linear and cubic interpolation. The *iws* parameter sets the number of input samples used for calculating one output sample (allowed values are any integer multiply of 4 in the range 4 - 1024); higher values mean better quality and slower speed.

The vdelayxw opcodes change the position of the write tap in the delay line (unlike all other delay ugens that move the read tap), and are most useful for implementing Doppler effects where the position of the listener is fixed, and the sound source is moving.

The multichannel opcodes (eg. *vdelayxq*) allow delaying 2 or 4 variables at once (stereo or quad signals); this is much more efficient than using separate opcodes for each channel.



### Notes

- Delay time is measured in seconds (unlike in *vdelay* and *vdelay3*), and must be a-rate.
- The minimum allowed delay is  $iws/2$  samples.
- Using the same variables as input and output is allowed in these opcodes.
- In *vdelayxw\**, changing the delay time has some effects on output volume:

$$a = 1 / (1 + dt)$$

where *a* is the output gain, and *dt* is the change of delay time per seconds.

- These opcodes are best used in the double-precision version of Csound.

## Examples

Here is an example of the use of the *vdelayxws* opcode. It uses the file *vdelayxws.csd* [examples/vdelayxws.csd].

### Example 1071. Example of the *vdelayxws* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o vdelayxws.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ims = .5      ;maximum delay time in seconds
iws = 1024    ;window size
adl = .5      ;delay time
asig1, asig2 diskin2 "kickroll.wav", 1, 0, 1 ;loop stereo file kickroll.wav
a2   poscil .2, .1, 1 ;make an LFO, 1 cycle per 10 seconds
adl = a2 + ims/2 ;offset the LFO so that it is positive
aoutL, aoutR vdelayxws asig1, asig2, adl, ims, iws ;use the LFO to control delay time
outs aoutL, aoutR

endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1

i 1 0 10

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*vdelayx*, *vdelayxq*, *vdelayxs*, *vdelayxw*, *vdelayxwq*

# vdelayk

vdelayk — k-rate variable time delay.

## Description

Variable delay applied to a k-rate signal

## Syntax

```
kout vdelayk ksig, kdel, imaxdel [, iskip, imode]
```

## Initialization

*imaxdel* - maximum value of delay in seconds.

*iskip* (optional) - Skip initialization if present and non zero.

*imode* (optional) - if non-zero it suppresses linear interpolation. While, normally, interpolation increases the quality of a signal, it should be suppressed if using vdelay with discrete control signals, such as, for example, trigger signals.

## Performance

*kout* - delayed output signal

*ksig* - input signal

*kdel* - delay time in seconds can be varied at k-rate

*vdelayk* is similar to *vdelay*, but works at k-rate. It is designed to delay control signals, to be used, for example, in algorithmic composition.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vdivv

vdivv — Performs division between two vectorial control signals

## Description

Performs division between two vectorial control signals

## Syntax

```
vdivv ifn1, ifn2, kelements [, kdstoffset] [, ksrcoffset] [,kverbose]
```

## Initialization

*ifn1* - number of the table hosting the first vector to be processed

*ifn2* - number of the table hosting the second vector to be processed

## Performance

*kelements* - number of elements of the two vectors

*kdstoffset* - index offset for the destination (*ifn1*) table (Default=0)

*ksrcoffset* - index offset for the source (*ifn2*) table (Default=0)

*kverbose* - Selects whether or not warnings are printed (Default=0)

*vdivv* divides two vectorial control signals, that is, each element of *ifn1* is divided by the corresponding element of *ifn2*. Each vectorial signal is hosted by a table (*ifn1* and *ifn2*). The number of elements contained in both vectors must be the same.

The result is a new vectorial control signal that overrides old values of *ifn1*. If you want to keep the old *ifn1* vector, use *vcopy\_i* opcode to copy it in another table. You can use *kdstoffset* and *ksrcoffset* to specify vectors in any location of the tables.

Negative values for *kdstoffset* and *ksrcoffset* are acceptable. If *kdstoffset* is negative, the out of range section of the vector will be discarded. If *ksrcoffset* is negative, the out of range elements will be assumed to be 0 (i.e. the destination elements will be set to 0). If elements for the destination vector are beyond the size of the table (including guard point), these elements are discarded (i.e. elements do not wrap around the tables). If elements for the source vector are beyond the table length, these elements are taken as 0 (i.e. the destination elements will be set to 0).

If the optional *kverbose* argument is different to 0, the opcode will print warning messages every k-pass if table lengths are exceeded.



### Warning

Using the same table as source and destination table in versions earlier than 5.04, might produce unexpected behavior, so use with care.

This opcode works at k-rate (this means that every k-pass the vectors are divided). There's an i-rate version of this opcode called *vdivv\_i*.



## Note

Please note that the *elements* argument has changed in version 5.03 from i-rate to k-rate. This will change the opcode's behavior in the unusual cases where the i-rate variable *ielements* is changed inside the instrument, for example in:

```

instr 1
ielements = 10
vadd 1, 1, ielements
ielements = 20
vadd 2, 1, ielements
turnoff
endin

```

All these operators (*vaddv*, *vsubv*, *vmultv*, *vdivv*, *vpowv*, *vexpv*, *vcopy* and *vmap*) are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2* etc.

## Examples

Here is an example of the *vdivv* opcode. It uses the file *vdivv.csd* [examples/vdivv.csd].

### Example 1072. Example of the *vdivv* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o cigoto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr=44100
ksmps=128
nchnls=2

instr 1
ifn1 = p4
ifn2 = p5
ielements = p6
idstoffset = p7
isrcoffset = p8
kval init 25
vdivv ifn1, ifn2, ielements, idstoffset, isrcoffset, 1
endin

instr 2 ;Printtable
itable = p4
isize = ftlen(itable)
kcount init 0
kval table kcount, itable
printk2 kval

if (kcount == isize) then
turnoff
endif

```

```
kcount = kcount + 1
  endin

</CsInstruments>
<CsScore>

f 1 0 16 -7 1 15 16

f 2 0 16 -7 1 15 2

i2 0.0 0.2 1
i2 0.2 0.2 2
i1 0.4 0.01 1 2 5 3 8
i2 0.8 0.2 1
i1 1.0 0.01 1 2 5 10 -2
i2 1.2 0.2 1
i1 1.4 0.01 1 2 8 14 0
i2 1.6 0.2 1
i1 1.8 0.01 1 2 8 0 14
i2 2.0 0.2 1
i1 2.2 0.002 1 1 8 5 2
i2 2.4 0.2 1
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado. Optional arguments added by Andres Cabrera and Istvan Varga.

New in Csound 5 (Previously available only on CsoundAV)

## vdivv\_i

`vdivv_i` — Performs division between two vectorial control signals at init time.

### Description

Performs division between two vectorial control signals at init time.

### Syntax

```
vdivv_i ifn1, ifn2, ielements [, idstoffset] [, isrcoffset]
```

### Initialization

*ifn1* - number of the table hosting the first vector to be processed

*ifn2* - number of the table hosting the second vector to be processed

*ielements* - number of elements of the two vectors

*idstoffset* - index offset for the destination (*ifn1*) table (Default=0)

*isrcoffset* - index offset for the source (*ifn2*) table (Default=0)

### Performance

`vdivv_i` divides two vectorial control signals, that is, each element of *ifn1* is divided by the corresponding element of *ifn2*. Each vectorial signal is hosted by a table (*ifn1* and *ifn2*). The number of elements contained in both vectors must be the same.

The result is a new vectorial control signal that overrides old values of *ifn1*. If you want to keep the old *ifn1* vector, use `vcopy_i` opcode to copy it in another table. You can use *idstoffset* and *isrcoffset* to specify vectors in any location of the tables.

Negative values for *idstoffset* and *isrcoffset* are acceptable. If *idstoffset* is negative, the out of range section of the vector will be discarded. If *isrcoffset* is negative, the out of range elements will be assumed to be 1 (i.e. the destination elements will not be changed). If elements for the destination vector are beyond the size of the table (including guard point), these elements are discarded (i.e. elements do not wrap around the tables). If elements for the source vector are beyond the table length, these elements are taken as 1 (i.e. the destination vector will not be changed for these elements).



#### Warning

Using the same table as source and destination table in versions earlier than 5.04, might produce unexpected behavior, so use with care.

This opcode works at init time. There's an k-rate version of this opcode called `vdivv`.

All these operators (`vaddv_i`, `vsubv_i`, `vmultv_i`, `vdivv_i`, `vpowv_i`, `vexpv_i`, `vcopy` and `vmap`) are designed to be used together with other opcodes that operate with vectorial signals such as `vcella`, `adsynt`, `adsynt2` etc.

## Credits

Written by Gabriel Maldonado. Optional arguments added by Andres Cabrera and Istvan Varga.

New in Csound 5 (Previously available only on CsoundAV)

# vecdelay

vecdelay — Vectorial Control-rate Delay Paths

## Description

Generate a sort of 'vectorial' delay

## Syntax

```
vecdelay ifn, ifnIn, ifnDel, ielements, imaxdel [, iskip]
```

## Initialization

*ifn* - number of the table containing the output vector

*ifnIn* - number of the table containing the input vector

*ifnDel* - number of the table containing a vector whose elements contain delay values in seconds

*ielements* - number of elements of the two vectors

*imaxdel* - Maximum value of delay in seconds.

*iskip* (optional) - initial disposition of delay-loop data space (see *reson*). The default value is 0.

## Performance

*vecdelay* is similar to *vdelay*, but it works at k-rate and, instead of delaying a single signal, it delays a vector. *ifnIn* is the input vector of signals, *ifn* is the output vector of signals, and *ifnDel* is a vector containing delay times for each element, expressed in seconds. Elements of *ifnDel* can be updated at k-rate. Each single delay can be different from that of the other elements, and can vary at k-rate. *imaxdel* sets the maximum delay allowed for all elements of *ifnDel*.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# veloc

veloc — Get the velocity from a MIDI event.

## Description

Get the velocity from a MIDI event.

## Syntax

```
ival veloc [ilow] [, ihigh]
```

## Initialization

*ilow*, *ihigh* -- low and hi ranges for mapping

## Performance

Get the MIDI byte value (0 - 127) denoting the velocity of the current event.

## Examples

Here is an example of the veloc opcode. It uses the file *veloc.csd* [examples/veloc.csd].

### Example 1073. Example of the veloc opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -+rtmidi=virtual -M0 ;;realtime audio I/O with MIDI in
;-iadc ;;uncomment -iadc if realtime audio input is needed too
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gisine ftgen 0, 0, 1024, 10, 1

instr 1

ivel veloc 0, 1 ;scale 0 - 1
print ivel ;print velocity
asig poscil .5*ivel, 220, gisine ;and use it as amplitude
outs asig, asig

endin
</CsInstruments>
<CsScore>
```

```
f 0 30 ;runs 30 seconds  
  
e  
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*aftouch, ampmidi, cpsmidi, cpsmidib, midictrl, notnum, octmidi, octmidib, pchbend, pchmidi, pchmidib*

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

# vexp

vexp — Performs power-of operations between a vector and a scalar

## Description

Performs power-of operations between a vector and a scalar

## Syntax

```
vexp ifn, kval, kelements [ , kdstoffset] [ , kverbose]
```

## Initialization

*ifn* - number of the table hosting the vectorial signal to be processed

## Performance

*kval* - scalar operand to be processed

*kelements* - number of elements of the vector

*kdstoffset* - index offset for the destination table (Optional, default = 0)

*kverbose* - Selects whether or not warnings are printed (Default=0)

*vexp* rises *kval* to each element contained in a vector from table *ifn*, starting from table index *kdstoffset*. This enables you to process a specific section of a table by specifying the offset and the number of elements to be processed. Offset is counted starting from 0, so if no offset is specified (or set to 0), the table will be modified from the beginning.

Note that this opcode runs at k-rate so the value of *kval* is processed every control period. Use with care or you will end up with very large (or small) numbers (or use *vexp\_i*).

These opcodes (*vadd*, *vmult*, *vpow* and *vexp*) perform numeric operations between a vectorial control signal (hosted by the table *ifn*), and a scalar signal (*kval*). Result is a new vector that overrides old values of *ifn*. All these opcodes work at k-rate.

Negative values for *kdstoffset* are valid. Elements from the vector that are outside the table, will be discarded, and they will not wrap around the table.

If the optional *kverbose* argument is different to 0, the opcode will print warning messages every k-pass if table lengths are exceeded.

In all these opcodes, the resulting vectors are stored in *ifn*, overriding the initial vectors. If you want to keep initial vector, use *vcopy* or *vcopy\_i* to copy it in another table. All these operators are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2*, etc. They can also be useful in conjunction with the spectral opcodes *pvsftw* and *pvsftr*.



### Note

Please note that the *elements* argument has changed in version 5.03 from i-rate to k-rate. This will change the opcode's behavior in the unusual cases where the i-rate variable *ielements* is changed inside the instrument, for example in:

```
instr 1
ielements = 10
vadd 1, 1, ielements
ielements = 20
vadd 2, 1, ielements
turnoff
endin
```

## Examples

Here is an example of the vexp opcode. It uses the file *vexp.csd* [examples/vexp.csd].

### Example 1074. Example of the vexp opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out Audio in
-odac -iadc ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o cigoto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr=44100
ksmps=128
nchnls=2

instr 1
ifn1 = p4
ival = p5
ielements = p6
idstoffset = p7
kval init 25
vexp ifn1, ival, ielements, idstoffset, 1
endin

instr 2 ;Printtable
itable = p4
isize = ftlen(itable)
kcount init 0
kval table kcount, itable
printk2 kval

if (kcount == isize) then
turnoff
endif

kcount = kcount + 1
endin

</CsInstruments>

<CsScore>

f 1 0 16 -7 1 16 17

i2 0.0 0.2 1
```

```
i1 0.4 0.01 1 2 3 4
i2 0.8 0.2 1
i1 1.0 0.01 1 0.5 5 -3
i2 1.2 0.2 1
i1 1.4 0.01 1 1.5 10 12
i2 1.6 0.2 1
e
```

```
</CsScore>
```

```
</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado. Optional arguments added by Andres Cabrera and Istvan Varga.

New in Csound 5 (Previously available only on CsoundAV)

# vexp\_i

vexp\_i — Performs power-of operations between a vector and a scalar

## Description

Performs power-of operations between a vector and a scalar

## Syntax

```
vexp_i ifn, ival, ielements[, idstoffset]
```

## Initialization

*ifn* - number of the table hosting the vectorial signal to be processed

*ival* - scalar operand to be processed

*ielements* - number of elements of the vector

*ival* - scalar value to be added

*idstoffset* - index offset for the destination table

## Performance

*vexp\_i* rises *ival* to each element contained in a vector from table *ifn*, starting from table index *idstoffset*. This enables you to process a specific section of a table by specifying the offset and the number of elements to be processed. Offset is counted starting from 0, so if no offset is specified (or set to 0), the table will be modified from the beginning.

Negative values for *idstoffset* are valid. Elements from the vector that are outside the table, will be discarded, and they will not wrap around the table.

This opcode runs only on initialization, there is a k-rate version of this opcode called *vexp*.

In all these opcodes, the resulting vectors are stored in *ifn*, overriding the initial vectors. If you want to keep initial vector, use *vcopy* or *vcopy\_i* to copy it in another table. All these operators are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2*, etc. They can also be useful in conjunction with the spectral opcodes *pvsftw* and *pvsftr*.

## Examples

Here is an example of the *vexp\_i* opcode. It uses the file *vexp\_i.csd* [examples/vexp\_i.csd].

### Example 1075. Example of the vexp\_i opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>
```

```

; Select audio/midi flags here according to platform
; Audio out   Audio in
-odac          -iadc          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o cigoto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr=44100
ksmps=128
nchnls=2

instr 1
ifn1 = p4
ival = p5
ielements = p6
idstoffset = p7
kval init 25
vexp_i ifn1, ival, ielements, idstoffset
endin

instr 2 ;Printtable
itable = p4
isize = ftlen(itable)
kcount init 0
kval table kcount, itable
printk2 kval

if (kcount == isize) then
turnoff
endif

kcount = kcount + 1
endin

</CsInstruments>

<CsScore>

f 1 0 16 -7 1 16 17

i2 0.0 0.2 1
i1 0.4 0.01 1 2 3 4
i2 0.8 0.2 1
i1 1.0 0.01 1 0.5 5 -3
i2 1.2 0.2 1
i1 1.4 0.01 1 1.5 10 12
i2 1.6 0.2 1
e

</CsScore>

</CsoundSynthesizer>

```

## See also

*vadd*, *vmult\_i*, *vpow\_i* and *vexp\_i*.

## Credits

Written by Gabriel Maldonado. Optional arguments added by Andres Cabrera and Istvan Varga.

New in Csound 5 (Previously available only on CsoundAV)

# vexpseg

vexpseg — Vectorial envelope generator

## Description

Generate exponential vectorial segments

## Syntax

```
vexpseg ifnout, ielements, ifn1, idur1, ifn2 [, idur2, ifn3 [...]]
```

## Initialization

*ifnout* - number of table hosting output vectorial signal

*ifn1* - starting vector

*ifn2, ifn3, etc.* - vector after *idurx* seconds

*idur1* - duration in seconds of first segment.

*idur2, idur3, etc.* - duration in seconds of subsequent segments.

*ielements* - number of elements of vectors.

## Performance

These opcodes are similar to *linseg* and *expseg*, but operate with vectorial signals instead of with scalar signals.

Output is a vectorial control signal hosted by *ifnout* (that must be previously allocated), while each break-point of the envelope is actually a vector of values. All break-points must contain the same number of elements (*ielements*).

All these operators are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2*, etc.

## Example

Here is an example of the vexpseg opcode. It uses the files *vexpseg.csd* [examples/vexpseg.csd].

### Example 1076. Example of the vexpseg opcode.

```
<CsoundSynthesizer>  
<CsOptions>  
-odac -B441 -b441  
</CsOptions>  
<CsInstruments>
```

```
sr=44100
ksmps=10
nchnls=2

gilen init 32

gitable1 ftgen 0, 0, gilen, 10, 1
gitable2 ftgen 0, 0, gilen, 10, 1

gitable3 ftgen 0, 0, gilen, -7, 30, gilen, 35
gitable4 ftgen 0, 0, gilen, -7, 400, gilen, 450
gitable5 ftgen 0, 0, gilen, -7, 5000, gilen, 5500

instr 1
vcopy gitable2, gitable1, gilen
turnoff
endin

instr 2
vexpseg gitable2, 16, gitable3, 2, gitable4, 2, gitable5
endin

instr 3
kcount init 0
if kcount < 16 then
  kval table kcount, gitable2
  printk 0,kval
  kcount = kcount +1
else
  turnoff
endif

endin

</CsInstruments>
<CsScore>
i1 0 1
s
i2 0 10
i3 0 1
i3 1 1
i3 1.5 1
i3 2 1
i3 2.5 1
i3 3 1
i3 3.5 1
i3 4 1
i3 4.5 1

</CsScore>
</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado.

Example by Andres Cabrera.

New in Csound 5 (Previously available only on CsoundAV)

# vexpv

vexpv — Performs exponential operations between two vectorial control signals

## Description

Performs exponential operations between two vectorial control signals

## Syntax

```
vexpv ifn1, ifn2, kelements [, kdstoffset] [, ksrcoffset] [,kverbose]
```

## Initialization

*ifn1* - number of the table hosting the first vector to be processed

*ifn2* - number of the table hosting the second vector to be processed

## Performance

*kelements* - number of elements of the two vectors

*kdstoffset* - index offset for the destination (*ifn1*) table (Default=0)

*ksrcoffset* - index offset for the source (*ifn2*) table (Default=0)

*kverbose* - Selects whether or not warnings are printed (Default=0)

*vexpv* elevates each element of *ifn2* to the corresponding element of *ifn1*. Each vectorial signal is hosted by a table (*ifn1* and *ifn2*). The number of elements contained in both vectors must be the same.

The result is a new vectorial control signal that overrides old values of *ifn1*. If you want to keep the old *ifn1* vector, use *vcopy\_i* opcode to copy it in another table. You can use *kdstoffset* and *ksrcoffset* to specify vectors in any location of the tables.

Negative values for *kdstoffset* and *ksrcoffset* are acceptable. If *kdstoffset* is negative, the out of range section of the vector will be discarded. If *ksrcoffset* is negative, the out of range elements will be assumed to be 0 (i.e. the destination elements will be set to 1). If elements for the destination vector are beyond the size of the table (including guard point), these elements are discarded (i.e. elements do not wrap around the tables). If elements for the source vector are beyond the table length, these elements are taken as 0 (i.e. the destination elements will be set to 1).

If the optional *kverbose* argument is different to 0, the opcode will print warning messages every k-pass if table lengths are exceeded.



### Warning

Using the same table as source and destination table in versions earlier than 5.04, might produce unexpected behavior, so use with care.

This opcode works at k-rate (this means that every k-pass the vectors are processed). There's an i-rate version of this opcode called *vexpv\_i*.



## Note

Please note that the *elements* argument has changed in version 5.03 from i-rate to k-rate. This will change the opcode's behavior in the unusual cases where the i-rate variable *ielements* is changed inside the instrument, for example in:

```

instr 1
ielements =      10
      vadd      1, 1, ielements
ielements =      20
      vadd      2, 1, ielements
      turnoff
endin

```

All these operators (*vaddv*, *vsubv*, *vmultv*, *vdivv*, *vpowv*, *vexpv*, *vcopy* and *vmap*) are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2* etc.

## Examples

Here is an example of the *vexpv* opcode. It uses the file *vexpv.csd* [examples/vexpv.csd].

### Example 1077. Example of the *vexpv* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o cigoto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr=44100
ksmps=128
nchnls=2

instr 1
ifn1 = p4
ifn2 = p5
ielements = p6
idstoffset = p7
isrcoffset = p8
kval init 25
vexpv ifn1, ifn2, ielements, idstoffset, isrcoffset, 1
endin

instr 2 ;Printtable
itable = p4
isize = ftlen(itable)
kcount init 0
kval table kcount, itable
printk2 kval

if (kcount == isize) then
  turnoff
endif

```

```
kcount = kcount + 1
endin

</CsInstruments>

<CsScore>

f 1 0 16 -7 1 16 17

f 2 0 16 -7 0 16 1

i2 0.0 0.2 1
i2 0.2 0.2 2
i1 0.4 0.01 1 2 5 3 8
i2 0.8 0.2 1
i1 1.0 0.01 1 2 5 10 -2
i2 1.2 0.2 1
i1 1.4 0.01 1 2 8 14 0
i2 1.6 0.2 1
i1 1.8 0.002 1 2 8 0 14
i2 2.0 0.2 1
i1 2.2 0.002 1 1 8 5 2
i2 2.4 0.2 1
e

</CsScore>

</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado. Optional arguments added by Andres Cabrera and Istvan Varga.

New in Csound 5 (Previously available only on CsoundAV)

# vexpv\_i

*vexpv\_i* — Performs exponential operations between two vectorial control signals at init time.

## Description

Performs exponential operations between two vectorial control signals at init time.

## Syntax

```
vexpv_i ifn1, ifn2, ielements [, idstoffset] [, isrcoffset]
```

## Initialization

*ifn1* - number of the table hosting the first vector to be processed

*ifn2* - number of the table hosting the second vector to be processed

*ielements* - number of elements of the two vectors

*idstoffset* - index offset for the destination (*ifn1*) table (Default=0)

*isrcoffset* - index offset for the source (*ifn2*) table (Default=0)

## Performance

*vexpv\_i* elevates each element of *ifn2* to the corresponding element of *ifn1*. Each vectorial signal is hosted by a table (*ifn1* and *ifn2*). The number of elements contained in both vectors must be the same.

The result is a new vectorial control signal that overrides old values of *ifn1*. If you want to keep the old *ifn1* vector, use *vcopy\_i* opcode to copy it in another table. You can use *idstoffset* and *isrcoffset* to specify vectors in any location of the tables.

Negative values for *idstoffset* and *isrcoffset* are acceptable. If *idstoffset* is negative, the out of range section of the vector will be discarded. If *isrcoffset* is negative, the out of range elements will be assumed to be 1 (i.e. the destination elements will be set to 1). If elements for the destination vector are beyond the size of the table (including guard point), these elements are discarded (i.e. elements do not wrap around the tables). If elements for the source vector are beyond the table length, these elements are taken as 1 (i.e. the destination vector elements will be set to 1).



### Warning

Using the same table as source and destination table in versions earlier than 5.04, might produce unexpected behavior, so use with care.

This opcode works at init time. There's an k-rate version of this opcode called *vexpv*.

All these operators (*vaddv\_i*, *vsubv\_i*, *vmultv\_i*, *vdivv\_i*, *vpowv\_i*, *vexpv\_i*, *vcopy* and *vmap*) are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2* etc.

## Credits

Written by Gabriel Maldonado. Optional arguments added by Andres Cabrera and Istvan Varga.

New in Csound 5 (Previously available only on CsoundAV)

# vibes

vibes — Physical model related to the striking of a metal block.

## Description

Audio output is a tone related to the striking of a metal block as found in a vibraphone. The method is a physical model developed from Perry Cook, but re-coded for Csound.

## Syntax

```
ares vibes kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivibfn, idec
```

## Initialization

*ihrd* -- the hardness of the stick used in the strike. A range of 0 to 1 is used. 0.5 is a suitable value.

*ipos* -- where the block is hit, in the range 0 to 1.

*imp* -- a table of the strike impulses. The file *marmstk1.wav* [examples/marmstk1.wav] is a suitable function from measurements and can be loaded with a *GENO1* table. It is also available at <ftp://ftp.c-s.bath.ac.uk/pub/dream/documentation/sounds/modelling/>.

*ivfn* -- shape of tremolo, usually a sine table, created by a function

*idec* -- time before end of note when damping is introduced

## Performance

*kamp* -- Amplitude of note.

*kfreq* -- Frequency of note played.

*kvibf* -- frequency of tremolo in Hertz. Suggested range is 0 to 12

*kvamp* -- amplitude of the tremolo

## Examples

Here is an example of the vibes opcode. It uses the file *vibes.csd* [examples/vibes.csd], and *marmstk1.wav* [examples/marmstk1.wav].

### Example 1078. Example of the vibes opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc      -d          ;;RT audio I/O
```

```
; For Non-realtime output leave only the line below:
; -o vibes.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
ksmps = 10
nchnls = 2

; Instrument #1.
instr 1
  ; kamp = 20000
  ; kfreq = 440
  ; ihrd = 0.5
  ; ipos = p4
  ; imp = 1
  ; kvibf = 6.0
  ; kvamp = 0.05
  ; ivibfn = 2
  ; idec = 0.1
  asig vibes 20000, 440, .5, p4 , 1, 6.0, 0.05, 2, .1
  outs asig, asig
endin

</CsInstruments>
<CsScore>

; Table #1, the "marmstkl.wav" audio file.
f 1 0 256 1 "marmstkl.wav" 0 0 0
; Table #2, a sine wave for the vibrato.
f 2 0 128 10 1

; Play Instrument #1 for four seconds.
i 1 0 4 0.561
i 1 + 4 1
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*marimba*

## Credits

Author: John ffitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

# vibr

vibr — Easier-to-use user-controllable vibrato.

## Description

Easier-to-use user-controllable vibrato.

## Syntax

```
kout vibr kAverageAmp, kAverageFreq, ifn
```

## Initialization

*ifn* -- Number of vibrato table. It normally contains a sine or a triangle wave.

## Performance

*kAverageAmp* -- Average amplitude value of vibrato

*kAverageFreq* -- Average frequency value of vibrato (in cps)

*vibr* is an easier-to-use version of *vibrato*. It has the same generation-engine of *vibrato*, but the parameters corresponding to missing input arguments are hard-coded to default values.

## Examples

Here is an example of the vibr opcode. It uses the file *vibr.csd* [examples/vibr.csd].

### Example 1079. Example of the vibr opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o vibr.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kaverageamp init 500
kaveragefreq init 4
kvib vibr kaverageamp, kaveragefreq, 1
```

```
asig poscil .8, 220+kvib, 1 ;add vibrato
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine wave

i 1 0 10

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*jitter, jitter2, vibrato*

## Credits

Author: Gabriel Maldonado

New in Version 4.15

# vibrato

vibrato — Generates a natural-sounding user-controllable vibrato.

## Description

Generates a natural-sounding user-controllable vibrato.

## Syntax

kout **vibrato** *kAverageAmp*, *kAverageFreq*, *kRandAmountAmp*, *kRandAmountFreq*, *kAmpMinRate*, *kAmpMaxRate*, *kcps*

## Initialization

*ifn* -- Number of vibrato table. It normally contains a sine or a triangle wave.

*iphs* -- (optional) Initial phase of table, expressed as a fraction of a cycle (0 to 1). A negative value will cause phase initialization to be skipped. The default value is 0.

## Performance

*kAverageAmp* -- Average amplitude value of vibrato

*kAverageFreq* -- Average frequency value of vibrato (in cps)

*kRandAmountAmp* -- Amount of random amplitude deviation

*kRandAmountFreq* -- Amount of random frequency deviation

*kAmpMinRate* -- Minimum frequency of random amplitude deviation segments (in cps)

*kAmpMaxRate* -- Maximum frequency of random amplitude deviation segments (in cps)

*kcpsMinRate* -- Minimum frequency of random frequency deviation segments (in cps)

*kcpsMaxRate* -- Maximum frequency of random frequency deviation segments (in cps)

*vibrato* outputs a natural-sounding user-controllable vibrato. The concept is to randomly vary both frequency and amplitude of the oscillator generating the vibrato, in order to simulate the irregularities of a real vibrato.

In order to have a total control of these random variations, several input arguments are present. Random variations are obtained by two separated segmented lines, the first controlling amplitude deviations, the second the frequency deviations. Average duration of each segment of each line can be shortened or enlarged by the arguments *kAmpMinRate*, *kAmpMaxRate*, *kcpsMinRate*, *kcpsMaxRate*, and the deviation from the average amplitude and frequency values can be independently adjusted by means of *kRandAmountAmp* and *kRandAmountFreq*.

## Examples

Here is an example of the vibrato opcode. It uses the file *vibrato.csd* [examples/vibrato.csd].

## Example 1080. Example of the vibrato opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o vibrato.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kaverageamp   init .5
kaveragefreq  init 5
krandamountamp line p4, p3, p5 ;increase random amplitude of vibrato
krandamountfreq init .3
kampminrate   init 3
kampmaxrate   init 5
kcpsminrate   init 3
kcpsmaxrate   init 5
kvib vibrato kaverageamp, kaveragefreq, krandamountamp, krandamountfreq, kampminrate, kampmaxrate, kcps
asig poscil .8, 220+kvib, 1 ;add vibrato
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine wave

i 1 0 15 .01 20

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*jitter, jitter2, vibr*

## Credits

Author: Gabriel Maldonado

New in Version 4.15

# vincr

vincr — Accumulates audio signals.

## Description

*vincr* increments one audio variable with another signal, i.e. it accumulates output.

## Syntax

```
vincr accum, aincr
```

## Performance

*accum* -- audio-rate accumulator variable to be incremented

*aincr* -- incrementing signal

*vincr* (variable increment) and *clear* are intended to be used together. *vincr* stores the result of the sum of two audio variables into the first variable itself (which is intended to be used as an accumulator in polyphony). The accumulator is typically a global variable that is used to combine signals from several sources (different instruments or instrument instances) for further processing (for example, via a global effect that reads the accumulator) or for outputting the combined signal by some means other than one of the *out* opcodes (eg. via the *fout* opcode). After the accumulator is used, the accumulator variable should be set to zero by means of the *clear* opcode (or it will explode).

## Examples

The following example uses the *vincr* opcode. It uses the file *vincr.csd* [examples/vincr.csd].

### Example 1081. Example of the *vincr* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o vincr.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gaReverb init 0

instr 1
```

```

idur = p3
kpitch = p4
a1 diskin2 "fox.wav", kpitch
a1 = a1*.5 ;reduce volume
vincr gaReverb, a1
endin

instr 99 ; global reverb
a1, ar reverbsc gaReverb, gaReverb, .8, 10000
outs gaReverb+a1, gaReverb+ar

clear gaReverb

endin

</CsInstruments>
<CsScore>

i1 0 3 1
i99 0 5
e

</CsScore>
</CsoundSynthesizer>

```

This is another example uses the vincr opcode. It uses the file *vincr-complex.csd* [examples/vincr-complex.csd].

### Example 1082.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac ;;realtime audio out
;-iadc ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o vincr-complex.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gaReverbSend init 0

instr 1

iamp = p4
ifreq = p5
aenv linseg 0.0, 0.1*p3, iamp, 0.6*p3, iamp, 0.3*p3, 0.0
aos poscil aenv, ifreq, 1
vincr gaReverbSend, aos
endin

instr 2 ; global reverb instrument

a1, ar reverbsc gaReverbSend, gaReverbSend, 0.85, 12000
outs gaReverbSend+a1, gaReverbSend+ar
clear gaReverbSend
endin

</CsInstruments>

```

```
<CsScore>
f1 0 4096 10 1

{ 4 CNT
  { 8 PARTIAL
    ; start time duration amplitude frequency
    i1 [0.5 * $CNT.] [1 + ($CNT * 0.2)] [.04 + (~ * .02)] [800 + (200 * $CNT.) + ($PARTIAL. * 20
  }
}

i2 0 6
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*clear*

## Credits

Author: Gabriel Maldonado  
Italy  
1999

New in Csound version 3.56

# vlimit

vlimit — Limiting and Wrapping Vectorial Signals

## Description

Limits elements of vectorial control signals.

## Syntax

```
vlimit ifn, kmin, kmax, ielements
```

## Initialization

*ifn* - number of the table hosting the vector to be processed

*ielements* - number of elements of the vector

## Performance

*kmin* - minimum threshold value

*kmax* - maximum threshold value

*vlimit* set lower and upper limits on each element of the vector they process.

These opcodes are similar to *limit*, *wrap* and *mirror*, but operate with a vectorial signal instead of with a scalar signal.

Result overrides old values of *ifn1*, if these are out of min/max interval. If you want to keep input vector, use *vcopy* opcode to copy it in another table.

All these opcodes are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2* etc.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vlinseg

vlinseg — Vectorial envelope generator

## Description

Generate linear vectorial segments

## Syntax

```
vlinseg ifnout, ielements, ifn1, idur1, ifn2 [, idur2, ifn3 [...]]
```

## Initialization

*ifnout* - number of table hosting output vectorial signal

*ifn1* - starting vector

*ifn2, ifn3, etc.* - vector after *idurx* seconds

*idur1* - duration in seconds of first segment.

*idur2, idur3, etc.* - duration in seconds of subsequent segments.

*ielements* - number of elements of vectors.

## Performance

These opcodes are similar to *linseg* and *expseg*, but operate with vectorial signals instead of with scalar signals.

Output is a vectorial control signal hosted by *ifnout* (that must be previously allocated), while each break-point of the envelope is actually a vector of values. All break-points must contain the same number of elements (*ielements*).

All these operators are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2*, etc.

## Example

Here is an example of the vlinseg opcode. It uses the files *vlinseg.csd* [examples/vlinseg.csd].

### Example 1083. Example of the vlinseg opcode.

```
<CsoundSynthesizer>  
<CsOptions>  
-odac -B441 -b441  
</CsOptions>  
<CsInstruments>
```

```
sr=44100
ksmps=10
nchnls=2

gilen init 32

gitable1 ftgen 0, 0, gilen, 10, 1
gitable2 ftgen 0, 0, gilen, 10, 1

gitable3 ftgen 0, 0, gilen, -7, 30, gilen, 35
gitable4 ftgen 0, 0, gilen, -7, 400, gilen, 450
gitable5 ftgen 0, 0, gilen, -7, 5000, gilen, 5500

instr 1
vcopy gitable2, gitable1, gilen
turnoff
endin

instr 2
vlinseg gitable2, 16, gitable3, 2, gitable4, 2, gitable5
endin

instr 3
kcount init 0
if kcount < 16 then
  kval table kcount, gitable2
  printk 0,kval
  kcount = kcount +1
else
  turnoff
endif

endin

</CsInstruments>
<CsScore>
i1 0 1
s
i2 0 10
i3 0 1
i3 1 1
i3 1.5 1
i3 2 1
i3 2.5 1
i3 3 1
i3 3.5 1
i3 4 1
i3 4.5 1

</CsScore>
</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado.

Example by Andres Cabrera.

New in Csound 5 (Previously available only on CsoundAV)

# vlowres

vlowres — A bank of filters in which the cutoff frequency can be separated under user control.

## Description

A bank of filters in which the cutoff frequency can be separated under user control

## Syntax

```
ares vlowres asig, kfco, kres, iord, ksep
```

## Initialization

*iord* -- total number of filters (1 to 10)

## Performance

*asig* -- input signal

*kfco* -- frequency cutoff (not in Hz)

*kres* -- resonance amount

*ksep* -- frequency cutoff separation for each filter: the first filter has a *kfco* cutoff, the second has a *kfco* + *ksep* and the third *kfco* + 2\**ksep* and so on, depending on the number of filters.

*vlowres* (variable resonant lowpass filter) allows a variable response curve in resonant filters. It can be thought of as a bank of lowpass resonant filters, each with the same resonance, serially connected. The frequency cutoff of each filter can vary with the *kfco* and *ksep* parameters.

## Examples

Here is an example of the vlowres opcode. It uses the file *vlowres.csd* [examples/vlowres.csd].

### Example 1084. Example of the vlowres opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o vlowres.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
```

```
Odbfs = 1

instr 1

kamp init p4
asig vco2 kamp, 110 ;saw wave
kfco line 30, p3, 300 ;vary the cutoff frequency from 30 to 300 Hz.
kres = 20
ksep = p5 ;different resonance values
iord = p6 ;and different number of filters
aout vlowres asig, kfco, kres, iord, ksep
aclip clip aout, 1, 1 ;avoid distortion
outs aclip, aclip

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine

s
i 1 0 10 .1 5 2 ;compensate volume and
i 1 + 10 .1 25 2 ;number of filters = 2
s
i 1 0 10 .01 5 6 ;compensate volume and
i 1 + 10 .04 15 6 ;number of filters = 6

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.49

## vmap

vmap — Maps elements from a vector according to indexes contained in another vector.

### Description

Maps elements from a vector onto another according to the indexes of a this vector.

### Syntax

```
vmap ifn1, ifn2, ielements [,idstoffset, isrcoffset]
```

### Initialization

*ifn1* - number of the table where the vectorial signal will be copied, and which contains the mapping vector

*ifn2* - number of the table hosting the vectorial signal to be copied

*ielements* - number of elements to process

*idstoffset* - index offset for destination table (*ifn1*)

*isrcoffset* - index offset for source table (*ifn2*)

### Performance

*vmap* maps elements of *ifn2* according to the values of table *ifn1*. Elements of *ifn1* are treated as indexes of table *ifn2*, so element values of *ifn1* must not exceed the length of *ifn2* table otherwise a Csound will report an error. Elements of *ifn1* are treated as integers, so any fractional part will be truncated. There is no interpolation performed on this operation.

In practice, what happens is that the elements of *ifn1* are used as indexes to *ifn2*, and then are replaced by the corresponding elements from *ifn2*. *ifn1* must be different from *ifn2*, otherwise the results are unpredictable. Csound will produce an init error if they are not.

All these operators (*vaddv*, *vsubv*, *vmultv*, *vdivv*, *vpowv*, *vexpv*, *vcopy* and *vmap*) are designed to be used together with other opcodes that operate with vectorial signals such as, *vcella*, *adsynt*, *adsynt2*, etc.

### Examples

Here is an example of the *vmap* opcode. It uses the file *vmap.csd* [examples/vmap.csd].

#### Example 1085. Example of the vmap opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>

<CsOptions>
; Select audio/midi flags here according to platform
```

```

; Audio out   Audio in
-odac         -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o vmap.wav -W ;; for file output any platform
</CsOptions>

<CsInstruments>
ksmps = 256
nchnls = 2
gisize = 64

gitable ftgen 0, 0, gisize, 10, 1 ;Table to be processed
gimap1 ftgen 0, 0, gisize, -7, gisize-1, gisize-1, 0 ; Mapping function to reverse table
gimap2 ftgen 0, 0, gisize, -5, 1, gisize-1, gisize-1 ; Mapping function for PWM
gimap3 ftgen 0, 0, gisize, -7, 1, (gisize/2)-1, gisize-1, 1, 1, (gisize/2)-1, gisize-1 ; Double frequ

instr 1 ;Hear an oscillator using gitable
asig oscil 10000, 440, gitable
outs asig,asig
endin

instr 2 ;Reverse the table (no sound change, except for a single click
vmap gimap1, gitable, gisize
vcopy_i gitable, gimap1, gisize
turnoff
endin

instr 3 ;Non-interpolated PWM (or phase waveshaping)
vmap gimap2, gitable, gisize
vcopy_i gitable, gimap2, gisize
turnoff
endin

instr 4 ;Double frequency
vmap gimap3, gitable, gisize
vcopy_i gitable, gimap3, gisize
turnoff
endin

</CsInstruments>

<CsScore>
i 1 0 8

i 2 2 1
i 3 4 1
i 4 6 1

e
</CsScore>

</CsoundSynthesizer>

```

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vmirror

vmirror — Limiting and Wrapping Vectorial Signals

## Description

'Reflects' elements of vectorial control signals on thresholds.

## Syntax

```
vmirror ifn, kmin, kmax, ielements
```

## Initialization

*ifn* - number of the table hosting the vector to be processed

*ielements* - number of elements of the vector

## Performance

*kmin* - minimum threshold value

*kmax* - maximum threshold value

*vmirror* 'reflects' each element of corresponding vector if it exceeds low or high thresholds.

These opcodes are similar to *limit*, *wrap* and *mirror*, but operate with a vectorial signal instead of with a scalar signal.

Result overrides old values of *ifn1*, if these are out of min/max interval. If you want to keep input vector, use *vcopy* opcode to copy it in another table.

All these opcodes are designed to be used together with other opcodes that operate with vectorial signals such as, *vcella*, *adsynt*, *adsynt2* etc.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vmult

vmult — Multiplies a vector in a table by a scalar value.

## Description

Multiplies a vector in a table by a scalar value.

## Syntax

```
vmult ifn, kval, kelements [, kdstoffset] [, kverbose]
```

## Initialization

*ifn* - number of the table hosting the vectorial signal to be processed

## Performance

*kval* - scalar value to be multiplied

*kelements* - number of elements of the vector

*kdstoffset* - index offset for the destination table (Optional, default = 0)

*kverbose* - Selects whether or not warnings are printed (Default=0)

*vmult* multiplies each element of the vector contained in the table *ifn* by *kval*, starting from table index *kdstoffset*. This enables you to process a specific section of a table by specifying the offset and the number of elements to be processed. Offset is counted starting from 0, so if no offset is specified (or set to 0), the table will be modified from the beginning.

Note that this opcode runs at k-rate so the value of *kval* is multiplied every control period. Use with care or you will end up with very large numbers (or use *vmult\_i*).

These opcodes (*vadd*, *vmult*, *vpow* and *vexp*) perform numeric operations between a vectorial control signal (hosted by the table *ifn*), and a scalar signal (*kval*). Result is a new vector that overrides old values of *ifn*. All these opcodes work at k-rate.

Negative values for *kdstoffset* are valid. Elements from the vector that are outside the table, will be discarded, and they will not wrap around the table.

If the optional *kverbose* argument is different to 0, the opcode will print warning messages every k-pass if table lengths are exceeded.

In all these opcodes, the resulting vectors are stored in *ifn*, overriding the initial vectors. If you want to keep initial vector, use *vcopy* or *vcopy\_i* to copy it in another table. All these operators are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2*, etc. They can also be useful in conjunction with the spectral opcodes *pvsftw* and *pvsftr*.



### Note

Please note that the *elements* argument has changed in version 5.03 from i-rate to k-rate. This will change the opcode's behavior in the unusual cases where the i-rate variable *ielements* is changed inside the instrument, for example in:

```

instr 1
ielements = 10
vadd 1, 1, ielements
ielements = 20
vadd 2, 1, ielements
turnoff
endin

```

## Example

Here is an example of the `vmult` opcode. It uses the file `vmult-2.csd` [examples/vmult-2.csd].

### Example 1086. Example of the `vmult` opcode.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o cigoto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr=44100
ksmps=128
nchnls=2

instr 1
ifn1 = p4
ival = p5
ielements = p6
idstoffset = p7
kval init 25
vmult ifn1, ival, ielements, idstoffset, 1
endin

instr 2 ;Printtable
itable = p4
isize = ftlen(itable)
kcount init 0
kval table kcount, itable
printk2 kval

if (kcount == isize) then
turnoff
endif

kcount = kcount + 1
endin

</CsInstruments>

<CsScore>

f 1 0 16 -7 1 16 17

i2 0.0 0.2 1
i1 0.4 0.01 1 2 3 4
i2 0.8 0.2 1
i1 1.0 0.01 1 0.5 5 -3

```

```

i2 1.2 0.2 1
i1 1.4 0.01 1 1.5 10 12
i2 1.6 0.2 1
e

```

```
</CsScore>
```

```
</CsoundSynthesizer>
```

Here is another example of the vmult opcode. It uses the file *vmult.csd* [examples/vmult.csd].

### Example 1087. Example of the vmult opcode.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc          ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o cigoto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr=44100
kr=4410
ksmps=10
nchnls=2

instr 1 ;table playback
ar lposcil 1, 1, 0, 262144, 1
outs ar,ar
endin

instr 2
vcopy 2, 1, 40000 ;copy vector from sample to empty table
vmult 5, 10000, 262144 ;scale noise to make it audible
vcopy 1, 5, 40000 ;put noise into sample
turnoff
endin

instr 3
vcopy 1, 2, 40000 ;put original information back in
turnoff
endin

</CsInstruments>
<CsScore>
f1 0 262144 -1 "beats.wav" 0 4 0
f2 0 262144 2 0

f5 0 262144 21 3 30000

i1 0 4
i2 3 1

s
i1 0 4
i3 3 1
s

i1 0 4

</CsScore>
</CsoundSynthesizer>

```

## See also

*vadd\_i*, *vadd\_i*, *vmult*, *vpow* and *vexp*.

## Credits

Written by Gabriel Maldonado. Optional arguments added by Andres Cabrera and Istvan Varga.

Example by Andres Cabrera.

New in Csound 5 (Previously available only on CsoundAV)

# vmult\_i

vmult\_i — Multiplies a vector in a table by a scalar value.

## Description

Multiplies a vector in a table by a scalar value.

## Syntax

```
vmult_i ifn, ival, ielements [, idstoffset]
```

## Initialization

*ifn* - number of the table hosting the vectorial signal to be processed

*ival* - scalar value to be multiplied

*ielements* - number of elements of the vector

*idstoffset* - index offset for the destination table

## Performance

*vmult\_i* multiplies each element of the vector contained in the table *ifn* by *ival*, starting from table index *idstoffset*. This enables you to process a specific section of a table by specifying the offset and the number of elements to be processed. Offset is counted starting from 0, so if no offset is specified (or set to 0), the table will be modified from the beginning.

This opcode runs only on initialization, there is a k-rate version of this opcode called *vmult*.

Negative values for *idstoffset* are valid. Elements from the vector that are outside the table, will be discarded, and they will not wrap around the table.

In all these opcodes, the resulting vectors are stored in *ifn*, overriding the initial vectors. If you want to keep initial vector, use *vcopy* or *vcopy\_i* to copy it in another table. All these operators are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2*, etc. They can also be useful in conjunction with the spectral opcodes *pvsftw* and *pvsftr*.

## Examples

Here is an example of the *vmult\_i* opcode. It uses the file *vmult\_i.csd* [examples/vmult\_i.csd].

### Example 1088. Example of the vmult\_i opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
; Audio out  Audio in
```

```
-odac          -iadc          ;;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o cigoto.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr=44100
ksmps=128
nchnls=2

instr 1
ifn1 = p4
ival = p5
ielements = p6
idstoffset = p7
kval init 25
vmult_i ifn1, ival, ielements, idstoffset
endin

instr 2 ;Printtable
itable = p4
isize = ftlen(itable)
kcount init 0
kval table kcount, itable
printk2 kval

if (kcount == isize) then
  turnoff
endif

kcount = kcount + 1
endin

</CsInstruments>

<CsScore>

f 1 0 16 -7 1 16 17

i2 0.0 0.2 1
i1 0.4 0.01 1 2 3 4
i2 0.8 0.2 1
i1 1.0 0.01 1 0.5 5 -3
i2 1.2 0.2 1
i1 1.4 0.01 1 1.5 10 12
i2 1.6 0.2 1
e

</CsScore>

</CsoundSynthesizer>
```

## See also

*vadd*, *vadd*, *vmult*, *vpow\_i*, *vpow*, *vexp\_i* and *vexp*.

## Credits

Written by Gabriel Maldonado. Optional arguments added by Andres Cabrera and Istvan Varga.

Example by Andres Cabrera.

New in Csound 5 (Previously available only on CsoundAV)

# vmultv

vmultv — Performs multiplication between two vectorial control signals

## Description

Performs multiplication between two vectorial control signals

## Syntax

```
vmultv ifn1, ifn2, kelements [, kdstoffset] [, ksrcoffset] [,kverbose]
```

## Initialization

*ifn1* - number of the table hosting the first vector to be processed

*ifn2* - number of the table hosting the second vector to be processed

## Performance

*kelements* - number of elements of the two vectors

*kdstoffset* - index offset for the destination (*ifn1*) table (Default=0)

*ksrcoffset* - index offset for the source (*ifn2*) table (Default=0)

*kverbose* - Selects whether or not warnings are printed (Default=0)

*vmultv* multiplies two vectorial control signals, that is, each element of the first vector is processed (only) with the corresponding element of the other vector. Each vectorial signal is hosted by a table (*ifn1* and *ifn2*). The number of elements contained in both vectors must be the same.

The Result is a new vectorial control signal that overrides old values of *ifn1*. If you want to keep the old *ifn1* vector, use *vcopy\_i* opcode to copy it in another table. You can use *kdstoffset* and *ksrcoffset* to specify vectors in any location of the tables.

Negative values for *kdstoffset* and *ksrcoffset* are acceptable. If *kdstoffset* is negative, the out of range section of the vector will be discarded. If *ksrcoffset* is negative, the out of range elements will be assumed to be 1 (i.e. the destination elements will not be changed). If elements for the destination vector are beyond the size of the table (including guard point), these elements are discarded (i.e. elements do not wrap around the tables). If elements for the source vector are beyond the table length, these elements are taken as 1 (i.e. the destination vector will not be changed for these elements).

If the optional *kverbose* argument is different to 0, the opcode will print warning messages every k-pass if table lengths are exceeded.



### Warning

Using the same table as source and destination table in versions earlier than 5.04, might produce unexpected behavior, so use with care.

This opcode works at k-rate (this means that every k-pass the vectors are multiplied). There's an i-rate version of this opcode called *vmultv\_i*.



## Note

Please note that the *elements* argument has changed in version 5.03 from i-rate to k-rate. This will change the opcode's behavior in the unusual cases where the i-rate variable *ielements* is changed inside the instrument, for example in:

```

instr 1
ielements =      10
      vadd      1, 1, ielements
ielements =      20
      vadd      2, 1, ielements
      turnoff
endin

```

All these operators (*vaddv*, *vsubv*, *vmultv*, *vdivv*, *vpowv*, *vexpv*, *vcopy* and *vmap*) are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2* etc.

## Examples

Here is an example of the *vmultv* opcode. It uses the file *vmultv.csd* [examples/vmultv.csd].

### Example 1089. Example of the *vmultv* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o cigoto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr=44100
ksmps=128
nchnls=2

instr 1
ifn1 = p4
ifn2 = p5
ielements = p6
idstoffset = p7
isrcoffset = p8
kval init 25
vmultv ifn1, ifn2, ielements, idstoffset, isrcoffset, 1
endin

instr 2 ;Printtable
itable = p4
isize = ftlen(itable)
kcount init 0
kval table kcount, itable
printk2 kval

if (kcount == isize) then
  turnoff
endif

```

```
kcount = kcount + 1
endin

</CsInstruments>

<CsScore>

f 1 0 16 -7 1 16 17

f 2 0 16 -7 1 16 2

i2 0.0 0.2 1
i2 0.2 0.2 2
i1 0.4 0.01 1 2 5 3 8
i2 0.8 0.2 1
i1 1.0 0.01 1 2 5 10 -2
i2 1.2 0.2 1
i1 1.4 0.01 1 2 8 14 0
i2 1.6 0.2 1
i1 1.8 0.01 1 2 8 0 14
i2 2.0 0.2 1
i1 2.2 0.002 1 1 8 5 2
i2 2.4 0.2 1
e

</CsScore>

</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado. Optional arguments added by Andres Cabrera and Istvan Varga.

New in Csound 5 (Previously available only on CsoundAV)

## vmultv\_i

`vmultv_i` — Performs multiplication between two vectorial control signals at init time.

### Description

Performs multiplication between two vectorial control signals at init time.

### Syntax

```
vmultv_i ifn1, ifn2, ielements [, idstoffset] [, isrcoffset]
```

### Initialization

*ifn1* - number of the table hosting the first vector to be processed

*ifn2* - number of the table hosting the second vector to be processed

*ielements* - number of elements of the two vectors

*idstoffset* - index offset for the destination (*ifn1*) table (Default=0)

*isrcoffset* - index offset for the source (*ifn2*) table (Default=0)

### Performance

`vmultv_i` multiplies two vectorial control signals, that is, each element of the first vector is processed (only) with the corresponding element of the other vector. Each vectorial signal is hosted by a table (*ifn1* and *ifn2*). The number of elements contained in both vectors must be the same.

The result is a new vectorial control signal that overrides old values of *ifn1*. If you want to keep the old *ifn1* vector, use `vcopy_i` opcode to copy it in another table. You can use *idstoffset* and *isrcoffset* to specify vectors in any location of the tables.

Negative values for *idstoffset* and *isrcoffset* are acceptable. If *idstoffset* is negative, the out of range section of the vector will be discarded. If *isrcoffset* is negative, the out of range elements will be assumed to be 1 (i.e. the destination elements will not be changed). If elements for the destination vector are beyond the size of the table (including guard point), these elements are discarded (i.e. elements do not wrap around the tables). If elements for the source vector are beyond the table length, these elements are taken as 1 (i.e. the destination vector will not be changed for these elements).



#### Warning

Using the same table as source and destination table in versions earlier than 5.04, might produce unexpected behavior, so use with care.

This opcode works at init time. There's an k-rate version of this opcode called `vmultv`.

All these operators (`vaddv_i`, `vsubv_i`, `vmultv_i`, `vdivv_i`, `vpowv_i`, `vexpv_i`, `vcopy` and `vmap`) are designed to be used together with other opcodes that operate with vectorial signals such as `vcella`, `adsynt`, `adsynt2` etc.

## Credits

Written by Gabriel Maldonado. Optional arguments added by Andres Cabrera and Istvan Varga.

New in Csound 5 (Previously available only on CsoundAV)

# voice

voice — An emulation of a human voice.

## Description

An emulation of a human voice.

## Syntax

ares **voice** *kamp*, *kfreq*, *kphoneme*, *kform*, *kvibf*, *kvamp*, *ifn*, *ivfn*

## Initialization

*ifn*, *ivfn* -- two table numbers containing the carrier waveform and the vibrato waveform. The files *impuls20.aiff* [examples/impuls20.aiff], *ahh.aiff* [examples/ahh.aiff], *eee.aiff* [examples/eee.aiff], or *ooo.aiff* [examples/ooo.aiff] are suitable for the first of these, and a sine wave for the second. These files are available from <ftp://ftp.cs.bath.ac.uk/pub/dream/documentation/sounds/modelling/>.

## Performance

*kamp* -- Amplitude of note.

*kfreq* -- Frequency of note played. It can be varied in performance.

*kphoneme* -- an integer in the range 0 to 16, which select the formants for the sounds:

- “eee”, “ihh”, “ehh”, “aaa”,
- “ahh”, “aww”, “ohh”, “uhh”,
- “uuu”, “ooo”, “rrr”, “lll”,
- “mmm”, “nnn”, “nng”, “ngg”.

At present the phonemes

- “fff”, “sss”, “thh”, “shh”,
- “xxx”, “hee”, “hoo”, “hah”,
- “bbb”, “ddd”, “jjj”, “ggg”,
- “vvv”, “zzz”, “thz”, “zhh”

are not available (!)

*kform* -- Gain on the phoneme. values 0.0 to 1.2 recommended.

*kvibf* -- frequency of vibrato in Hertz. Suggested range is 0 to 12

*kvamp* -- amplitude of the vibrato

## Examples

Here is an example of the voice opcode. It uses the file *voice.csd* [examples/voice.csd], and *impuls20.aiff* [examples/impuls20.aiff].

### Example 1090. Example of the voice opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o voice.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kamp = p4
kphon = p5
asig voice kamp, 200, kphon, 0.488, 0, 1, 1, 2
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 256 1 "impuls20.aiff" 0 0 0 ;audio file for the carrier waveform
f 2 0 256 10 1 ;sine wave for the vibrato waveform

;      ampl phoneme
i 1 0 2 0.8 1
i 1 + . 0.6 2
i 1 + . 1.8 3
i 1 + . 15.0 4
i 1 + . 0.05 5
i 1 + . 0.06 6
i 1 + . 0.03 7
i 1 + . 0.0002 8
i 1 + . 0.1 9
i 1 + . 0.5 10
i 1 + . 100 11
i 1 + . 0.03 12
i 1 + . 0.04 13
i 1 + . 0.04 14
i 1 + . 0.04 15
i 1 + . 0.05 16

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitich (after Perry Cook)

University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

# vosim

vosim — Simple vocal simulation based on glottal pulses with formant characteristics.

## Description

This opcode produces a simple vocal simulation based on glottal pulses with formant characteristics. Output is a series of sound events, where each event is composed of a burst of squared sine pulses followed by silence. The VOSIM (VOcal SIMulation) synthesis method was developed by Kaegi and Tempelaars in the 1970's.

## Syntax

```
ar vosim kamp, kFund, kForm, kDecay, kPulseCount, kPulseFactor, ifn [, iskip]
```

## Intialization

*ifn* - a sound table, normally containing half a period of a sinewave, squared (see notes below).

*iskip* - (optional) Skip initialization, for tied notes.

## Performance

*ar* - output signal. Note that the output is usually unipolar - positive only.

*kamp* - output amplitude, the peak amplitude of the first pulse in each burst.

*kFund* - fundamental pitch, in Herz. Each event is 1/kFund seconds long.

*kForm* - formant center frequency. Length of each pulse in the burst is 1/kForm seconds.

*kDecay* - a dampening factor from pulse to pulse. This is subtracted from amplitude on each new pulse.

*kPulseCount* - number of pulses in the burst part of each event.

*kPulseFactor* - the pulse width is multiplied by this value at each new pulse. This results in formant sweeping. If factor is < 1.0, the formant sweeps up, if > 1.0 each new pulse is longer, so the formant sweeps down. The final pitch of the formant is  $kForm * \text{pow}(kPulseFactor, kPulseCount)$

The output of *vosim* is a series of sound events, where each event is composed of a burst of squared sine pulses followed by silence. The total duration of the events determines fundamental frequency. The length of each single pulse in the squared-sine bursts produce a formant frequency band. The width of the formant is determined by rate of silence to pulses (see below). The final result is also shaped by the dampening factor from pulse to pulse.

A small practical problem in using this opcode is that no GEN function will create a squared sine wave out of the box. Something like the following can be used to create the appropriate table from the score.

```
; use GEN09 to create half a sine in table 17
f 17 time size 9 0.5 1 0

; run instr 101 on table 17 for a single init-pass
i 101 0 0 17
```

It can also be done with an instrument writing to an f-table in the orchestra:

```

; square each point in table #p4. This should be run as init-only, just once in the performance.

instr 101

    index tablen p4

    index = index - 1 ; start from last point
loop:

    ival table index, p4

    ival = ival * ival

    tableiw ival, index, p4

    index = index - 1

    if index < 0 igoto endloop
        igoto loop
endloop:
endin

```



## Parameter Limits

The count of pulses multiplied by pulse width should fit in the event length ( $1/kFund$ ). If this is not fulfilled, the algorithm does not break, we just do not start any pulses that would outlast the event. This might introduce a silence at end of event even if none was intended. In consequence,  $kForm$  should be higher than  $kFund$ , otherwise only silence is output.

*Vosim* was created to emulate voice sounds using a model of glottal pulse. Rich sounds can be created by combining several instances of *vosim* with different parameters. One drawback is that the signal is not band-limited. But as the authors point out, attenuation of high-pitch components is -60 dB at 6 times the fundamental frequency. The signal can also be changed by changing the source signal in the lookup table. The technique has historical interest, and can produce rich sound very cheaply (each sample requires only a table lookup and a single multiplication for attenuation).

As stated, formant bandwidth depends on the ratio between pulse burst and silence in an event. But this is not an independent parameter: The fundamental decides event length, and formant center defines the pulse length. It is therefore impossible to guarantee a specific burst/silence ratio, since the burst length has to be an integer multiple of pulse length. The decay of pulses can be used to smooth the transition from  $N$  to  $N\pm 1$  pulses, but there will still be steps in the spectral profile of output. The example code below shows one approach to this.

All input parameters are k-rate. The input parameters are only used to set up each new event (or grain). Event amplitude is fixed for each event at initialization. In normal parameter ranges, when  $ksmps < 500$ , the k-rate parameters are updated more often than events are created. In any case, no wide-band noise will be injected in the system due to k-rate inputs being updated less often than they are read, but some other artefacts could be created.

The opcode should behave reasonably in the face of all user inputs. Some details:

- $kFund < 0$ : This is forced to positive - no point in "reversed" events.
- $kFund == 0$ : This leads to "infinite" length event, ie a pulse burst followed by very long indefinite silence.
- $kForm == 0$ : This leads to infinite length pulse, so no pulses are generated (i.e. silence).

- d.  $kForm < 0$ : Table is read backward. If table is symmetric,  $kform$  and  $-kform$  should give bit-identical outputs.
- e.  $kPulseFactor == 0$ : Second pulse onwards is zero. See (c).
- f.  $kPulseFactor < 0$ : Pulses alternately read table forward and reversed.

With asymmetric pulse table there may be some use for negative  $kForm$  or negative  $kPulseFactor$ .

## Examples

Here is an example of the `vosim` opcode. It uses the file `vosim.csd` [examples/vosim.csd].

### Example 1091. Example of the `vosim` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out   Audio in
-odac          -iadc    ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o vosim.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
sr          = 44100
ksmps      = 100
nchnls     = 1

;#####
; By Rasmus Ekman 2008

; Square each point in table #p4. This should only be run once in the performance.
instr 10

  index tableng p4
  index = index - 1 ; start from last point
loop:
  ival table index, p4
  ival = ival * ival
  tableiw ival, index, p4
  index = index - 1
  if index < 0 igoto endloop
  igoto loop
endloop:
endin

;#####

; Main vosim instrument. Sweeps from a fund1/form1 to fund2/form2,
; trying for narrowest formant bandwidth (still quite wide by the looks of it)
; p4:      amp
; p5, p6: fund beg-end
; p7, p8: form beg-end
; p9:      amp decay (ignored)
; p10:     pulse count (ignored - calc internally)
; p11:     pulse length mod
; p12:     skip (for tied events)
; p13:     don't fade out (if followed by tied note)
instr 1
  kamp    init p4
          ; freq start, end
```

```

kfund line p5, p3, p6
; formant start, end
kform line p7, p3, p8

; Try for constant ratio burst/silence, and narrowest formant bandwidth
kPulseCount = (kform / kfund) ;init p10
; Attempt to smooth steps between format bandwidths,
; increasing decay before we are forced to a lower pulse count
kDecay = kPulseCount/(kform % kfund) ; init p9
if (kDecay * kPulseCount) > kamp then
  kDecay = kamp / kPulseCount
endif
kDecay = 0.3 * kDecay

kPulseFactor init p11

; ar vosim kamp, kFund, kForm, kDecay, kPulseCount, kPulseFactor, ifn [, iskip]
  ar1 vosim kamp, kfund, kform, kDecay, kPulseCount, kPulseFactor, 17, p12

; scale amplitude for 16-bit files, with quick fade out
amp init 20000
if (p13 != 0) goto nofade
amp linseg 20000, p3-.02, 20000, .02, 0
nofade:
out ar1 * amp
endin

</CsInstruments>
<CsScore>

f1      0 32768  9 1  1 0 ; sine wave
f17     0 32768  9 0.5 1 0 ; half sine wave
i10 0 0 17 ; init run only, square table 17

; Vosim score

; Picking some formants from the table in Csound manual

;      p4=amp fund      form      decay pulses pulsemod [skip] nofade
; tenor a -> e
i1 0 .5 .5 280 240 650 400 .03 5 1
i1 . . .3 . . 1080 1700 .03 5 .
i1 . . .2 . . 2650 2600 .03 5 .
i1 . . .15 . . 2900 3200 .03 5 .

; tenor a -> o
i1 0.6 .2 .5 300 210 650 400 .03 5 1 0 1
i1 . . .3 . . 1080 800 .03 5 . . .
i1 . . .2 . . 2650 2600 .03 5 . . .
i1 . . .15 . . 2900 2800 .03 5 . . .

; tenor o -> aah
i1 .8 .3 .5 210 180 400 650 .03 5 1 1 1
i1 . . .3 . . 800 1080 .03 5 . . .
i1 . . .2 . . 2600 2650 .03 5 . . .
i1 . . .15 . . 2800 2900 .03 5 . . .

; tenor aa -> i
i1 1.1 .2 .5 180 250 650 290 .03 5 1 1 1
i1 . . .3 . . 1080 1870 .03 5 . . .
i1 . . .2 . . 2650 2800 .03 5 . . .
i1 . . .15 . . 2900 3250 .03 5 . . .

; tenor i -> u
i1 1.3 .3 .5 250 270 290 350 .03 5 1 1 0
i1 . . .3 . . 1870 600 .03 5 . . .
i1 . . .2 . . 2800 2700 .03 5 . . .
i1 . . .15 . . 3250 2900 .03 5 . . .

```

e

```
</CsScore>  
</CsoundSynthesizer>
```

## See also

*fof, fof2*

## Credits

Author: Rasmus Ekman  
March 2008

# vphaseseg

vphaseseg — Allows one-dimensional HVS (Hyper-Vectorial Synthesis).

## Description

*vphaseseg* allows one-dimensional HVS (Hyper-Vectorial Synthesis).

## Syntax

```
vphaseseg kphase, ioutab, ielems, itab1, idist1, itab2 \  
[ , idist2, itab3, ... , idistN-1, itabN]
```

## Initialization

*ioutab* - number of output table.

*ielem* - number of elements to process

*itab1*, ..., *itabN* - breakpoint table numbers

*idist1*, ..., *idistN-1* - distances between breakpoints in percentage values

## Performance

*kphase* - phase pointer

*vphaseseg* returns the coordinates of section points of an N-dimensional space path. The coordinates of section points are stored into an output table. The number of dimensions of the N-dimensional space is determined by the *ielem* argument that is equal to N and can be set to any number. To define the path, user have to provide a set of points of the N-dimensional space, called break-points. Coordinates of each break-point must be contained by a different table. The number of coordinates to insert in each break-point table must obviously equal to *ielem* argument. There can be any number of break-point tables filled by the user.

Hyper-Vectorial Synthesis actually deals with two kinds of spaces. The first space is the N-dimensional space in which the path is defined, this space is called time-variant parameter space (or SPACE A). The path belonging to this space is covered by moving a point into the second space that normally has a number of dimensions smaller than the first. Actually, the point in motion is the projection of corresponding point of the N-dimensional space (could also be considered a section of the path). The second space is called user-pointer-motion space (or SPACE B) and, in the case of *vphaseseg* opcode, has only ONE DIMENSION. Space B is covered by means of *kphase* argument (that is a sort of path pointer), and its range is 0 to 1. The output corresponding to current pointer value is stored in *ioutab* table, whose data can be afterwards used to control any synthesis parameters.

In *vphaseseg*, each break-point is separated from the other by a distance expressed in percentage, where all the path length is equal to the sum of all distances. So distances between breakpoints can be different, differently from kinds of HVS in which space B has more than one dimension, in these cases distance between break-points MUST be THE SAME for all intervals.

## See Also

*hvs1*, *hvs2*, *hvs3*

## Credits

Author: Gabriel Maldonado

New in version 5.06

# vport

vport — Vectorial Control-rate Delay Paths

## Description

Generate a sort of 'vectorial' portamento

## Syntax

```
vport ifn, khtime, ielements [, ifnInit]
```

## Initialization

*ifn* - number of the table containing the output vector

*ielements* - number of elements of the two vectors

*ifnInit* (optional) - number of the table containing a vector whose elements contain initial portamento values.

## Performance

*vport* is similar to *port*, but operates with vectorial signals, instead of with scalar signals. Each vector element is treated as an independent control signal. Input and output vectors are placed in the same table and output vector overrides input vector. If you want to keep input vector, use *vcopy* opcode to copy it in another table.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vpow

vpow — Raises each element of a vector to a scalar power.

## Description

Raises each element of a vector to a scalar power.

## Syntax

```
vpow ifn, kval, kelements [ , kdstoffset] [ , kverbose]
```

## Initialization

*ifn* - number of the table hosting the vectorial signal to be processed

## Performance

*kval* - scalar value to which the elements of *ifn* will be raised

*kelements* - number of elements of the vector

*kdstoffset* - index offset for the destination table (Optional, default = 0)

*kverbose* - Selects whether or not warnings are printed (Default=0)

*vpow* raises each element of the vector contained in the table *ifn* to the power of *kval*, starting from table index *kdstoffset*. This enables you to process a specific section of a table by specifying the offset and the number of elements to be processed. Offset is counted starting from 0, so if no offset is specified (or set to 0), the table will be modified from the beginning.

Note that this opcode runs at k-rate so the value of *kval* is processed every control period. Use with care or you will end up with very large (or small) numbers (or use *vpow\_i*).

These opcodes (*vadd*, *vmult*, *vpow* and *vexp*) perform numeric operations between a vectorial control signal (hosted by the table *ifn*), and a scalar signal (*kval*). Result is a new vector that overrides old values of *ifn*. All these opcodes work at k-rate.

Negative values for *kdstoffset* are valid. Elements from the vector that are outside the table, will be discarded, and they will not wrap around the table.

If the optional *kverbose* argument is different to 0, the opcode will print warning messages every k-pass if table lengths are exceeded.

In all these opcodes, the resulting vectors are stored in *ifn*, overriding the initial vectors. If you want to keep initial vector, use *vcopy* or *vcopy\_i* to copy it in another table. All these operators are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2*, etc. They can also be useful in conjunction with the spectral opcodes *pvsftw* and *pvsftr*.



### Note

Please note that the *elements* argument has changed in version 5.03 from i-rate to k-rate. This will change the opcode's behavior in the unusual cases where the i-rate variable *ielements* is changed inside the instrument, for example in:

```
instr 1
ielements = 10
vadd 1, 1, ielements
ielements = 20
vadd 2, 1, ielements
turnoff
endin
```

## Examples

Here is an example of the vpow opcode. It uses the file *vpow.csd* [examples/vpow.csd].

### Example 1092. Examples of the vpow opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out Audio in
-odac -iadc ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o cigoto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr=44100
ksmps=128
nchnls=2

instr 1
ifn1 = p4
ival = p5
ielements = p6
idstoffset = p7
kval init 25
vpow ifn1, ival, ielements, idstoffset, 1
endin

instr 2 ;Printtable
itable = p4
isize = ftlen(itable)
kcount init 0
kval table kcount, itable
printk2 kval

if (kcount == isize) then
turnoff
endif

kcount = kcount + 1
endin

</CsInstruments>

<CsScore>

f 1 0 16 -7 1 16 17

i2 0.0 0.2 1
```

```

i1 0.4 0.01 1 2 3 4
i2 0.8 0.2 1
i1 1.0 0.01 1 0.5 5 -3
i2 1.2 0.2 1
i1 1.4 0.01 1 1.5 10 12
i2 1.6 0.2 1
e

```

```
</CsScore>
```

```
</CsoundSynthesizer>
```

Here is another example of the `vpow` opcode. It uses the file `vpow-2.csd` [examples/vpow-2.csd].

### Example 1093.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o vpow-2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ain   diskin2 "fox.wav", 1   ;soundfile
fsrc  pvsanal ain, 1024, 256, 1024, 1
ifn   ftgen 0, 0, 1024/2, 2, 0   ;create empty function table for the 513 bins
kflag pvsftw fsrc,ifn   ;export only amplitudes to table
kval  line .001, p3, 1   ;start with big distortion, cahnge over note duration to clean sound
kbin  line p4, p3, p5   ;vary the bins
vpow  ifn, kval, kbin, 0   ;note that this operation is applied each k-cycle!
;vpow ifn, kval, kbin, 10   ;if you set kdstoffset to 10 it will affect bins 10+(kbin line p4, p3, p5)
pvsftr fsrc,ifn   ;read modified data back to fsrc
aout  pvsynth fsrc   ;and resynth
outs  aout*p6, aout*p6   ;adjust volume to compensate

endin
</CsInstruments>
<CsScore>

i 1 0 4 100 100 .02 ;first 100 bins are affected
i 1 + 4 10 10 .1 ;first 10 bins
i 1 + 4 1 400 .05 ;sweep from bin 1 to 400
e
</CsScore>
</CsoundSynthesizer>

```

## See also

`vadd_i`, `vmult`, `vpow` and `vexp`.

## Credits

Written by Gabriel Maldonado. Optional arguments added by Andres Cabrera and Istvan Varga.

New in Csound 5 (Previously available only on CsoundAV)

# vpow\_i

vpow\_i — Raises each element of a vector to a scalar power

## Description

Raises each element of a vector to a scalar power

## Syntax

```
vpow_i ifn, ival, ielements [, idstoffset]
```

## Initialization

*ifn* - number of the table hosting the vectorial signal to be processed

*ielements* - number of elements of the vector

*ival* - scalar value to which the elements of *ifn* will be raised

*idstoffset* - index offset for the destination table

## Performance

*vpow\_i* elevates each element of the vector contained in the table *ifn* to the power of *ival*, starting from table index *idstoffset*. This enables you to process a specific section of a table by specifying the offset and the number of elements to be processed. Offset is counted starting from 0, so if no offset is specified (or set to 0), the table will be modified from the beginning.

This opcode runs only on initialization, there is a k-rate version of this opcode called *vpow*.

Negative values for *idstoffset* are valid. Elements from the vector that are outside the table, will be discarded, and they will not wrap around the table.

In all these opcodes, the resulting vectors are stored in *ifn*, overriding the initial vectors. If you want to keep initial vector, use *vcopy* or *vcopy\_i* to copy it in another table. All these operators are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2*, etc. They can also be useful in conjunction with the spectral opcodes *pvsftw* and *pvsftr*.

## Examples

Here is an example of the *vpow\_i* opcode. It uses the file *vpow\_i.csd* [examples/vpow\_i.csd].

### Example 1094. Examples of the *vpow\_i* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform
```

```

; Audio out   Audio in
-odac         -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o cigoto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr=44100
ksmps=128
nchnls=2

instr 1
ifn1 = p4
ival = p5
ielements = p6
idstoffset = p7
kval init 25
vpow_i ifn1, ival, ielements, idstoffset
endin

instr 2 ;Printtable
itable = p4
isize = ftlen(itable)
kcount init 0
kval table kcount, itable
printk2 kval

if (kcount == isize) then
  turnoff
endif

kcount = kcount + 1
endin

</CsInstruments>

<CsScore>

f 1 0 16 -7 1 16 17

i2 0.0 0.2 1
i1 0.4 0.01 1 2 3 4
i2 0.8 0.2 1
i1 1.0 0.01 1 0.5 5 -3
i2 1.2 0.2 1
i1 1.4 0.01 1 1.5 10 12
i2 1.6 0.2 1
e

</CsScore>

</CsoundSynthesizer>

```

Here is another example of the `vpow_i` opcode. It uses the file `vpow_i-2.csd` [examples/vpow\_i-2.csd].

### Example 1095.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac ;;realtime audio out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o vpow_i-2.wav -W ;; for file output any platform

```

```
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gifil ftgen 1, 0, 0, 1, "fox.wav", 0, 0, 1

instr 1

ival      = p4      ;different distortion settings
ielements = p5
idstoffset = p6      ;index offset
vpow_i 1, ival, ielements, idstoffset
asig lposcil 1, 1, 0, 0, 1
      outs asig, asig

endin
</CsInstruments>
<CsScore>

i1 0 2.7 .5 70000 0 ;no offset
i1 3 2.7 .01 50000 70000 ;add another period of distortion, starting at sample 70000

e
</CsScore>
</CsoundSynthesizer>
```

## See also

*vadd\_i*, *vmult\_i*, *vpow* and *vexp\_i*.

## Credits

Written by Gabriel Maldonado. Optional arguments added by Andres Cabrera and Istvan Varga.

New in Csound 5 (Previously available only on CsoundAV)

# vpowv

vpowv — Performs power-of operations between two vectorial control signals

## Description

Performs power-of operations between two vectorial control signals

## Syntax

```
vpowv ifn1, ifn2, kelements [, kdstoffset] [, ksrcoffset] [,kverbose]
```

## Initialization

*ifn1* - number of the table hosting the first vector to be processed

*ifn2* - number of the table hosting the second vector to be processed

## Performance

*kelements* - number of elements of the two vectors

*kdstoffset* - index offset for the destination (*ifn1*) table (Default=0)

*ksrcoffset* - index offset for the source (*ifn2*) table (Default=0)

*kverbose* - Selects whether or not warnings are printed (Default=0)

*vpowv* raises each element of *ifn1* to the corresponding element of *ifn2*. Each vectorial signal is hosted by a table (*ifn1* and *ifn2*). The number of elements contained in both vectors must be the same.

The result is a new vectorial control signal that overrides old values of *ifn1*. If you want to keep the old *ifn1* vector, use *vcopy\_i* opcode to copy it in another table. You can use *kdstoffset* and *ksrcoffset* to specify vectors in any location of the tables.

Negative values for *kdstoffset* and *ksrcoffset* are acceptable. If *kdstoffset* is negative, the out of range section of the vector will be discarded. If *ksrcoffset* is negative, the out of range elements will be assumed to be 1 (i.e. the destination elements will not be changed). If elements for the destination vector are beyond the size of the table (including guard point), these elements are discarded (i.e. elements do not wrap around the tables). If elements for the source vector are beyond the table length, these elements are taken as 1 (i.e. the destination vector will not be changed for these elements).

If the optional *kverbose* argument is different to 0, the opcode will print warning messages every k-pass if table lengths are exceeded.



### Warning

Using the same table as source and destination table in versions earlier than 5.04, might produce unexpected behavior, so use with care.

This opcode works at k-rate (this means that every k-pass the vectors are processed). There's an i-rate version of this opcode called *vpowv\_i*.



## Note

Please note that the *elements* argument has changed in version 5.03 from i-rate to k-rate. This will change the opcode's behavior in the unusual cases where the i-rate variable *ielements* is changed inside the instrument, for example in:

```

instr 1
ielements =      10
      vadd      1, 1, ielements
ielements =      20
      vadd      2, 1, ielements
      turnoff
endin

```

All these operators (*vaddv*, *vsubv*, *vmultv*, *vdivv*, *vpowv*, *vexpv*, *vcopy* and *vmap*) are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2*, etc.

## Examples

Here is an example of the *vpowv* opcode. It uses the file *vpowv.csd* [examples/vpowv.csd].

### Example 1096. Example of the *vpowv* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o cigoto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr=44100
ksmps=128
nchnls=2

instr 1
ifn1 = p4
ifn2 = p5
ielements = p6
idstoffset = p7
isrcoffset = p8
kval init 25
vpowv ifn1, ifn2, ielements, idstoffset, isrcoffset, 1
endin

instr 2 ;Printtable
itable = p4
isize = ftlen(itable)
kcount init 0
kval table kcount, itable
printk2 kval

if (kcount == isize) then
  turnoff
endif

```

```
kcount = kcount + 1
endin

</CsInstruments>

<CsScore>

f 1 0 16 -7 1 16 17

f 2 0 16 -7 1 16 2

i2 0.0 0.2 1
i2 0.2 0.2 2
i1 0.4 0.01 1 2 5 3 8
i2 0.8 0.2 1
i1 1.0 0.01 1 2 5 10 -2
i2 1.2 0.2 1
i1 1.4 0.01 1 2 8 14 0
i2 1.6 0.2 1
i1 1.8 0.01 1 2 8 0 14
i2 2.0 0.2 1
e

</CsScore>

</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado. Optional arguments added by Andres Cabrera and Istvan Varga.

New in Csound 5 (Previously available only on CsoundAV)

# vpowv\_i

`vpowv_i` — Performs power-of operations between two vectorial control signals at init time.

## Description

Performs power-of operations between two vectorial control signals at init time.

## Syntax

```
vpowv_i ifn1, ifn2, ielements [, idstoffset] [, isrcoffset]
```

## Initialization

*ifn1* - number of the table hosting the first vector to be processed

*ifn2* - number of the table hosting the second vector to be processed

*ielements* - number of elements of the two vectors

*idstoffset* - index offset for the destination (*ifn1*) table

*isrcoffset* - index offset for the source (*ifn2*) table

## Performance

`vpowv_i` raises each element of *ifn1* to the corresponding element of *ifn2*. Each vectorial signal is hosted by a table (*ifn1* and *ifn2*). The number of elements contained in both vectors must be the same.

The result is a new vectorial control signal that overrides old values of *ifn1*. If you want to keep the old *ifn1* vector, use `vcopy_i` opcode to copy it in another table. You can use *idstoffset* and *isrcoffset* to specify vectors in any location of the tables.

Negative values for *idstoffset* and *isrcoffset* are acceptable. If *idstoffset* is negative, the out of range section of the vector will be discarded. If *isrcoffset* is negative, the out of range elements will be assumed to be 1 (i.e. the destination elements will not be changed). If elements for the destination vector are beyond the size of the table (including guard point), these elements are discarded (i.e. elements do not wrap around the tables). If elements for the source vector are beyond the table length, these elements are taken as 1 (i.e. the destination vector will not be changed for these elements).



### Warning

Using the same table as source and destination table in versions earlier than 5.04, might produce unexpected behavior, so use with care.

This opcode works at init time. There's an k-rate version of this opcode called `vpowv`.

All these operators (`vaddv_i`, `vsubv_i`, `vmultv_i`, `vdivv_i`, `vpowv_i`, `vexpv_i`, `vcopy` and `vmap`) are designed to be used together with other opcodes that operate with vectorial signals such as `vcella`, `adsynt`, `adsynt2` etc.

## Credits

Written by Gabriel Maldonado. Optional arguments added by Andres Cabrera and Istvan Varga.

New in Csound 5 (Previously available only on CsoundAV)

## vpvoc

`vpvoc` — Implements signal reconstruction using an fft-based phase vocoder and an extra envelope.

### Description

Implements signal reconstruction using an fft-based phase vocoder and an extra envelope.

### Syntax

```
ares vpvoc ktimepnt, kfmod, ifile [, ispecwp] [, ifn]
```

### Initialization

*ifile* -- the pvoc number (n in pvoc.n) or the name in quotes of the analysis file made using *pvanal*. (See *pvoc*.)

*ispecwp* (optional, default=0) -- if non-zero, attempts to preserve the spectral envelope while its frequency content is varied by *kfmod*. The default value is zero.

*ifn* (optional, default=0) -- optional function table containing control information for *vpvoc*. If *ifn* = 0, control is derived internally from a previous *tableseg* or *tablexseg* unit. Default is 0. (New in Csound version 3.59)

### Performance

*ktimepnt* -- The passage of time, in seconds, through the analysis file. *ktimepnt* must always be positive, but can move forwards or backwards in time, be stationary or discontinuous, as a pointer into the analysis file.

*kfmod* -- a control-rate transposition factor: a value of 1 incurs no transposition, 1.5 transposes up a perfect fifth, and .5 down an octave.

This implementation of *pvoc* was originally written by Dan Ellis. It is based in part on the system of Mark Dolson, but the pre-analysis concept is new. The spectral extraction and amplitude gating (new in Csound version 3.56) were added by Richard Karpen based on functions in SoundHack by Tom Erbe.

*vpvoc* is identical to *pvoc* except that it takes the result of a previous *tableseg* or *tablexseg* and uses the resulting function table (passed internally to the *vpvoc*), as an envelope over the magnitudes of the analysis data channels. Optionally, a table specified by *ifn* may be used.

The result is spectral enveloping. The function size used in the *tableseg* should be *framesize/2*, where *framesize* is the number of bins in the phase vocoder analysis file that is being used by the *vpvoc*. Each location in the table will be used to scale a single analysis bin. By using different functions for *ifn1*, *ifn2*, etc.. in the *tableseg*, the spectral envelope becomes a dynamically changing one. See also *tableseg* and *tablexseg*.

### Examples

The following example, using *vpvoc*, shows the use of functions such as

```
f 1 0 256 5 .001 128 1 128 .001
f 2 0 256 5 1 128 .001 128 1
f 3 0 256 7 1 256 1
```

to scale the amplitudes of the separate analysis bins.

```
ktime line 0, p3, 3 ; time pointer, in seconds, into file
tablexseg 1, p3*0.5, 2, p3*0.5, 3
apv vpvoc ktime, 1, "pvoc.file"
```

The result would be a time-varying “spectral envelope” applied to the phase vocoder analysis data. Since this amplifies or attenuates the amount of signal at the frequencies that are paired with the amplitudes which are scaled by these functions, it has the effect of applying very accurate filters to the signal. In this example the first table would have the effect of a band-pass filter, gradually be band-rejected over half the note’s duration, and then go towards no modification of the magnitudes over the second half.

Here is a complete example of the `vpvoc` opcode. It uses the file `vpvoc.csd` [examples/vpvoc.csd].

### Example 1097.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o vpvoc.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
; analyze "fox.wav" with PVANAL first
iend = p4
ktime line 0, p3, iend
tablexseg p5, p3, p6 ;morph from table 1
asig vpvoc ktime, 1, "fox.pvx" ;to table 2
outs asig*3, asig*3

endin
</CsInstruments>
<CsScore>
f 1 0 512 9 .5 1 0
f 2 0 512 5 1 60 0.01 390 0.01 62 1

i 1 0 5 2.7 1 2
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*pvoc*

## Credits

Authors: Dan Ellis and Richard Karpen  
Seattle, WA USA  
1997

New in version 3.44

# vrandh

vrandh — Generates a vector of random numbers stored into a table, holding the values for a period of time.

## Description

Generates a vector of random numbers stored into a table, holding the values for a period of time. Generates a sort of 'vectorial band-limited noise'.

## Syntax

```
vrandh ifn, krange, kcps, ielements [, idstoffset] [, iseed] \  
      [, isize] [, ioffset]
```

## Initialization

*ifn* - number of the table where the vectorial signal will be generated

*ielements* - number of elements of the vector

*idstoffset* - (optional, default=0) -- index offset for the destination table

*iseed* (optional, default=0.5) -- seed value for the recursive pseudo-random formula. A value between 0 and +1 will produce an initial output of  $kamp * iseed$ . A negative value will cause seed re-initialization to be skipped. A value greater than 1 will seed from system time, this is the best option to generate a different random sequence for each run.

*isize* (optional, default=0) -- if zero, a 16 bit number is generated. If non-zero, a 31-bit random number is generated. Default is 0.

*ioffset* - (optional, default=0) -- a base value added to the random result.

## Performance

*krange* - range of random elements (from *-krange* to *krange*).

*kcps* - rate of generated elements in cycles per seconds.

This opcode is similar to *randh*, but operates on vectors instead of with scalar values.

Though the argument *isize* defaults to 0, thus using a 16-bit random number generator, using the newer 31-bit algorithm is recommended, as this will produce a random sequence with a longer period (more random numbers before the sequence starts repeating).

The output is a vector contained in *ifn* (that must be previously allocated).

All these operators are designed to be used together with other opcodes that operate with vector such as *adsynt*, etc.

## Examples

Here is an example of the *vrandh* opcode. It uses the file *vrandh.csd* [examples/vrandh.csd].

**Example 1098. Example of the vrandh opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o vranh.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
;Example by Andres Cabrera

sr=44100
ksmps=128
nchnls=2

gitab ftgen 0, 0, 16, -7, 0, 128, 0

instr 1
  krange init p4
  kcps init p5
  ioffset init p6

  kav1 init 0
  kav2 init 0
  kcount init 0

;      table  krange kcps  ielements  idstoffset  iseed  isize ioffset
vrandh  gitab,  krange, kcps,      3,          3,          2,  0,  ioffset

  kfreq1 table 3, gitab
  kfreq2 table 4, gitab
  kfreq3 table 5, gitab

;Change the frequency of three oscillators according to the random values
aosc1 oscili 4000, kfreq1, 1
aosc2 oscili 2000, kfreq2, 1
aosc3 oscili 4000, kfreq3, 1

outs aosc1+aosc2, aosc3+aosc2
endin

</CsInstruments>
<CsScore>
f 1 0 1024 10 1
;      krange kcps  ioffset
i 1 0 5 100 1 300
i 1 5 5 300 1 400
i 1 10 5 100 2 1000
i 1 15 5 400 4 1000
i 1 20 5 1000 8 2000
i 1 25 5 250 16 300
e

</CsScore>
</CsoundSynthesizer>

```

## See also

*vrandi*, *randh*

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vrandi

vrandi — Generate a sort of 'vectorial band-limited noise'

## Description

Generate a sort of 'vectorial band-limited noise'

## Syntax

```
vrandi ifn, krange, kcps, ielements [, idstoffset] [, iseed] \  
      [, isize] [, ioffset]
```

## Initialization

*ifn* - number of the table where the vectorial signal will be generated

*ielements* - number of elements to process

*idstoffset* - (optional, default=0) -- index offset for the destination table

*iseed* (optional, default=0.5) -- seed value for the recursive pseudo-random formula. A value between 0 and +1 will produce an initial output of  $kamp * iseed$ . A negative value will cause seed re-initialization to be skipped. A value greater than 1 will seed from system time, this is the best option to generate a different random sequence for each run.

*isize* (optional, default=0) -- if zero, a 16 bit number is generated. If non-zero, a 31-bit random number is generated. Default is 0.

*ioffset* - (optional, default=0) -- a base value added to the random result.

## Performance

*krange* - range of random elements (from *-krange* to *krange*)

*kcps* - rate of generated elements in cycles per seconds

This opcode is similar to *randi*, but operates on vectors instead of with scalar values.

Though argument *isize* defaults to 0, thus using a 16-bit random number generator, using the newer 31-bit algorithm is recommended, as this will produce a random sequence with a longer period (more random numbers before the sequence starts repeating).

The output is a vector contained in *ifn* (that must be previously allocated).

All these operators are designed to be used together with other opcodes that operate with vector such as *adsynt*, etc.

## Examples

Here is an example of the vrandi opcode. It uses the file *vrandi.csd* [examples/vrandi.csd].

**Example 1099. Example of the vrandi opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc    -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o vrandi.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr=44100
ksmps=128
nchnls=2

;Example by Andres Cabrera

gitab ftgen 0, 0, 16, -7, 0, 128, 0

instr 1
  krange init p4
  kcps   init p5
  ioffset init p6
  ;      table  krange kcps  ielements  idstoffset  iseed  isize  ioffset
  vrandi gitab, krange, kcps,    3,        3,        2,    1,  ioffset

  kfreq1 table 3, gitab
  kfreq2 table 4, gitab
  kfreq3 table 5, gitab

  ;Change the frequency of three oscillators according to the random values
  aosc1 oscili 4000, kfreq1, 1
  aosc2 oscili 2000, kfreq2, 1
  aosc3 oscili 4000, kfreq3, 1

  outs aosc1+aosc2, aosc3+aosc2
endin

</CsInstruments>
<CsScore>

f 1 0 2048 10 1

;          krange kcps  ioffset
i 1 0 5 100 1 300
i 1 5 5 5 1 400
i 1 10 5 100 2 1000
i 1 15 5 400 4 1000
i 1 20 5 1000 8 2000
i 1 20 5 300 32 350

e

</CsScore>
</CsoundSynthesizer>

```

**See also**

*vrandh*, *randi*

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vstaudio, vstaudiog

vstaudio — VST audio output.

## Syntax

```
aout1,aout2 vstaudio instance, [ain1, ain2]
```

```
aout1,aout2 vstaudiog instance, [ain1, ain2]
```

## Description

*vstaudio* and *vstaudiog* are used for sending and receiving audio from a VST plugin.

*vstaudio* is used within an instrument definition that contains a *vstmidiout* or *vstnote* opcode. It outputs audio for only that one instrument. Any audio remaining in the plugin after the end of the note, for example a reverb tail, will be cut off and should be dealt with using a damping envelope.

*vstaudiog* (*vstaudio* global) is used in a separate instrument to process audio from any number of VST notes or MIDI events that share the same VST plugin instance (*instance*). The *vstaudiog* instrument must be numbered higher than all the instruments receiving notes or MIDI data, and the note controlling the *vstplug* instrument must have an indefinite duration, or at least a duration as long as the VST plugin is active.

## Initialization

*instance* - the number which identifies the plugin, to be passed to other *vst4cs* opcodes.

## Performance

*aout1*, *aout2* - the audio output received from the plugin.

*ain1*, *ain2* - the audio input sent to the plugin.

## Examples

Here is an example of the use of the *vstaudio* opcode. It uses the file *vst4cs.csd* [examples/vst4cs.csd].

### Example 1100. Example of the *vstaudio* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Credits: Adapted by Michael Gogins
; from code by David Horowitz and Lian Cheung.
; The "--displays" option is required in order for
; the Pianoteq GUI to dispatch events and display properly.
-m3 --displays -odac
</CsOptions>
<CsInstruments>
sr      = 44100
ksmps   = 20
```

```

nchnls = 2
gipianoteq ; Load the Pianoteq into memory.
vstinit    "C:\\Program Files\\Steinberg\\VstPlugins\\Pianoteq 3.0 Trial\\Pianoteq

; Print information about the Pianoteq, such as parameter names and numbers.
vstinfo    gipianoteq

; Open the Pianoteq's GUI.
vstedit    gipianoteq

; Send notes from the score to the Pianoteq.
instr 1
; MIDI channels are numbered starting at 0.
; p3 always contains the duration of the note.
; p4 contains the MIDI key number (pitch),
; p5 contains the MIDI velocity number (loudness),
imidichannel init          0
vstnote     gipianoteq, imidichannel, p4, p5, p3
endin

; Send parameter changes to the Pianoteq.
instr 2
; p4 is the parameter number.
; p5 is the parameter value.
vstparamset gipianoteq, p4, p5
endin

; Send audio from the Pianoteq to the output.
instr 3
ablankinput init          0
aleft, aright vstaudio     gipianoteq, ablankinput, ablankinput
outs        aleft, aright
endin

</CsInstruments>
<CsScore>
; Turn on the instrument that receives audio from the Pianoteq indefinitely.
i 3 0 -1
; Send parameter changes to Pianoteq before sending any notes.
; NOTE: All parameters must be between 0.0 and 1.0.
; Length of piano strings:
i 2 0 1 33 0.5
; Hammer noise:
i 2 0 1 25 0.1
; Send a C major 7th arpeggio to the Pianoteq.
i 1 1 10 60 76
i 1 2 10 64 73
i 1 3 10 67 70
i 1 4 10 71 67
; End the performance, leaving some time
; for the Pianoteq to finish sending out its audio,
; or for the user to play with the Pianoteq virtual keyboard.
e 20
</CsScore>
</CsoundSynthesizer>

```

## Credits

By: Andrés Cabrera and Michael Gogins

Uses code from Hermann Seib's VSTHost and Thomas Grill's vst~ object.

VST is a trademark of Steinberg Media Technologies GmbH. VST Plug-In Technology by Steinberg.

# vstbankload

vstbankload — Loads parameter banks to a VST plugin.

## Syntax

```
vstbankload instance, ipath
```

## Description

*vstbankload* is used for loading parameter banks to a VST plugin.

## Initialization

*instance* - the number which identifies the plugin, to be passed to other vst4cs opcodes.

*ipath* - the full pathname of the parameter bank (. fxb file).

## Examples

### Example 1101. Example for vstbankload

```
/* orc */
sr      = 44100
kr      = 4410
ksmps   = 10
nchnls  = 2

gihandle1 vstinit "c:/vstplugins/cheeze/cheeze machine.dll", 1

instr 4
  vstbankload gihandle1,"c:/vstplugins/cheeze/chengo'scheese.fxb"
  vstinfo     gihandle1
endin

/* sco */
i 3 0 21
i 4 1 1 57 32
```

## Credits

By: Andrés Cabrera and Michael Gogins

Uses code from Hermann Seib's VSTHost and Thomas Grill's vst~ object.

VST is a trademark of Steinberg Media Technologies GmbH. VST Plug-In Technology by Steinberg.

# vstedit

vstedit — Opens the GUI editor window for a VST plugin.

## Syntax

```
vstedit instance
```

## Description

*vstedit* opens the custom GUI editor window for a VST plugin. Note that not all VST plugins have custom GUI editors. It may be necessary to use the `--displays` command-line option to ensure that Csound handles events from the editor window and displays it properly.

## Initialization

*instance* - the number which identifies the plugin, to be passed to other *vst4cs* opcodes.

## Examples

Here is an example of the use of the *vstedit* opcode. It uses the file *vst4cs.csd* [examples/vst4cs.csd].

### Example 1102. Example of the *vstedit* opcode

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Credits: Adapted by Michael Gogins
; from code by David Horowitz and Lian Cheung.
; The "--displays" option is required in order for
; the Pianoteq GUI to dispatch events and display properly.
-m3 --displays -odac
</CsOptions>
<CsInstruments>
sr      = 44100
ksmps  = 20
nchnls = 2

; Load the Pianoteq into memory.
gipianoteq vstinit      "C:\\Program Files\\Steinberg\\VstPlugins\\Pianoteq 3.0 Trial\\Pianoteq

; Print information about the Pianoteq, such as parameter names and numbers.
vstinfo      gipianoteq

; Open the Pianoteq's GUI.
vstedit      gipianoteq

; Send notes from the score to the Pianoteq.
instr 1
; MIDI channels are numbered starting at 0.
; p3 always contains the duration of the note.
; p4 contains the MIDI key number (pitch),
; p5 contains the MIDI velocity number (loudness),
imidichannel init      0
vstnote      gipianoteq, imidichannel, p4, p5, p3
```

```
        endin

        ; Send parameter changes to the Pianoteq.
        instr 2
        ; p4 is the parameter number.
        ; p5 is the parameter value.
        vstparamset    gipianoteq, p4, p5
        endin

        ; Send audio from the Pianoteq to the output.
        instr 3
ablankinput    init            0
aleft, aright  vstaudio        gipianoteq, ablankinput, ablankinput
                outs          aleft, aright
        endin

</CsInstruments>
<CsScore>
; Turn on the instrument that receives audio from the Pianoteq indefinitely.
i 3 0 -1
; Send parameter changes to Pianoteq before sending any notes.
; NOTE: All parameters must be between 0.0 and 1.0.
; Length of piano strings:
i 2 0 1 33 0.5
; Hammer noise:
i 2 0 1 25 0.1
; Send a C major 7th arpeggio to the Pianoteq.
i 1 1 10 60 76
i 1 2 10 64 73
i 1 3 10 67 70
i 1 4 10 71 67
; End the performance, leaving some time
; for the Pianoteq to finish sending out its audio,
; or for the user to play with the Pianoteq virtual keyboard.
e 20
</CsScore>
</CsoundSynthesizer>
```

## Credits

By: Andrés Cabrera and Michael Gogins

Uses code from Hermann Seib's VSTHost and Thomas Grill's vst~ object.

VST is a trademark of Steinberg Media Technologies GmbH. VST Plug-In Technology by Steinberg.

# vstinit

vstinit — Load a VST plugin into memory for use with the other vst4cs opcodes.

## Syntax

```
instance vstinit ilibrarypath [,iverbose]
```

## Description

*vstinit* is used to load a VST plugin into memory for use with the other vst4cs opcodes. Both VST effects and instruments (synthesizers) can be used.

## Initialization

*instance* - the number which identifies the plugin, to be passed to other vst4cs opcodes.

*ilibrarypath* - the full path to the vst plugin shared library (DLL, on Windows). Remember to use '/' instead of '\' as separator.

*iverbose* - show plugin information and parameters when loading.

## Examples

Here is an example of the use of the *vstinit* opcode. It uses the file *vst4cs.csd* [examples/vst4cs.csd].

### Example 1103. Example of the *vstinit* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Credits: Adapted by Michael Gogins
; from code by David Horowitz and Lian Cheung.
; The "--displays" option is required in order for
; the Pianoteq GUI to dispatch events and display properly.
-m3 --displays -odac
</CsOptions>
<CsInstruments>
sr      = 44100
ksmps   = 20
nchnls = 2
; Load the Pianoteq into memory.
gipianoteq vstinit      "C:\\Program Files\\Steinberg\\VstPlugins\\Pianoteq 3.0 Trial\\Pianoteq
; Print information about the Pianoteq, such as parameter names and numbers.
vstinfo    gipianoteq
; Open the Pianoteq's GUI.
vstedit    gipianoteq
; Send notes from the score to the Pianoteq.
instr 1
; MIDI channels are numbered starting at 0.
```

```

; p3 always contains the duration of the note.
; p4 contains the MIDI key number (pitch),
; p5 contains the MIDI velocity number (loudness),
imidichannel  init          0
vstnote       gipianoteq, imidichannel, p4, p5, p3
endin

; Send parameter changes to the Pianoteq.
instr 2
; p4 is the parameter number.
; p5 is the parameter value.
vstparamset   gipianoteq, p4, p5
endin

; Send audio from the Pianoteq to the output.
instr 3
ablankinput   init          0
aleft, aright vstaudio       gipianoteq, ablankinput, ablankinput
outs
aleft, aright
endin

</CsInstruments>
<CsScore>
; Turn on the instrument that receives audio from the Pianoteq indefinitely.
i 3 0 -1
; Send parameter changes to Pianoteq before sending any notes.
; NOTE: All parameters must be between 0.0 and 1.0.
; Length of piano strings:
i 2 0 1 33 0.5
; Hammer noise:
i 2 0 1 25 0.1
; Send a C major 7th arpeggio to the Pianoteq.
i 1 1 10 60 76
i 1 2 10 64 73
i 1 3 10 67 70
i 1 4 10 71 67
; End the performance, leaving some time
; for the Pianoteq to finish sending out its audio,
; or for the user to play with the Pianoteq virtual keyboard.
e 20
</CsScore>
</CsoundSynthesizer>

```

## Credits

By: Andrés Cabrera and Michael Gogins

Uses code from Hermann Seib's VSTHost and Thomas Grill's vst~ object.

VST is a trademark of Steinberg Media Technologies GmbH. VST Plug-In Technology by Steinberg.

# vstinfo

vstinfo — Displays the parameters and the programs of a VST plugin.

## Syntax

```
vstinfo instance
```

## Description

*vstinfo* displays the parameters and the programs of a VST plugin.

Note: The *verbose* flag in *vstinit* gives the same information as *vstinfo*. *vstinfo* is useful after loading parameter banks, or when the plugin changes parameters dynamically.

## Initialization

*instance* - the number which identifies the plugin, to be passed to other vst4cs opcodes.

## Examples

Here is an example of the use of the *vstinfo* opcode. It uses the file *vst4cs.csd* [examples/vst4cs.csd].

### Example 1104. Example of the *vstinfo* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Credits: Adapted by Michael Gogins
; from code by David Horowitz and Lian Cheung.
; The "--displays" option is required in order for
; the Pianoteq GUI to dispatch events and display properly.
-m3 --displays -odac
</CsOptions>
<CsInstruments>
sr      = 44100
ksmps  = 20
nchnls = 2

; Load the Pianoteq into memory.
gipianoteq vstinit      "C:\\Program Files\\Steinberg\\VstPlugins\\Pianoteq 3.0 Trial\\Pianoteq

; Print information about the Pianoteq, such as parameter names and numbers.
vstinfo      gipianoteq

; Open the Pianoteq's GUI.
vstedit      gipianoteq

; Send notes from the score to the Pianoteq.
instr 1
; MIDI channels are numbered starting at 0.
; p3 always contains the duration of the note.
; p4 contains the MIDI key number (pitch),
; p5 contains the MIDI velocity number (loudness),
imidichannel init      0
```

```
vstnote          gipianoteq, imidichannel, p4, p5, p3
endin

; Send parameter changes to the Pianoteq.
instr 2
; p4 is the parameter number.
; p5 is the parameter value.
vstparamset     gipianoteq, p4, p5
endin

; Send audio from the Pianoteq to the output.
instr 3
ablaninput      init          0
aleft, aright   vstaudio      gipianoteq, ablaninput, ablaninput
outs          aleft, aright
endin

</CsInstruments>
<CsScore>
; Turn on the instrument that receives audio from the Pianoteq indefinitely.
i 3 0 -1
; Send parameter changes to Pianoteq before sending any notes.
; NOTE: All parameters must be between 0.0 and 1.0.
; Length of piano strings:
i 2 0 1 33 0.5
; Hammer noise:
i 2 0 1 25 0.1
; Send a C major 7th arpeggio to the Pianoteq.
i 1 1 10 60 76
i 1 2 10 64 73
i 1 3 10 67 70
i 1 4 10 71 67
; End the performance, leaving some time
; for the Pianoteq to finish sending out its audio,
; or for the user to play with the Pianoteq virtual keyboard.
e 20
</CsScore>
</CsoundSynthesizer>
```

## Credits

By: Andrés Cabrera and Michael Gogins

Uses code from Hermann Seib's VSTHost and Thomas Grill's vst~ object.

VST is a trademark of Steinberg Media Technologies GmbH. VST Plug-In Technology by Steinberg.

# vstmidiout

vstmidiout — Sends MIDI information to a VST plugin.

## Syntax

```
vstmidiout instance, kstatus, kchan, kdata1, kdata2
```

## Description

*vstmidiout* is used for sending MIDI information to a VST plugin.

## Initialization

*instance* - the number which identifies the plugin, to be passed to other vst4cs opcodes.

## Performance

*kstatus* - the type of midi message to be sent. Currently noteon (144), note off (128), Control Change (176), Program change (192), Aftertouch (208) and Pitch Bend (224) are supported.

*kchan* - the MIDI channel transmitted on.

*kdata1*, *kdata2* - the MIDI data pair, which varies depending on *kstatus*. e.g. note/velocity for note on and note off, Controller number/value for control change.

## Examples

### Example 1105. Example for vstmidiout

```
/* orc */
sr      = 44100
kr      = 4410
ksmps  = 10
nchnls = 2

gihandle1 vstinit "c:/vstplugins/cheeze/cheeze machine.dll", 1

instr 3
ain1    =          0
abl, ab2 vstaudio  gihandle1, ain1, ain1
          outs      abl, ab2
endin

instr 4
          vstmidiout gihandle1, 144, 1, p4, p5
endin

/* sco */
i 3 0 21

i 4 1 1 57 32
i 4 3 1 60 100
```

```
i 4 5 1 62 100  
i 4 7 1 64 100  
i 4 9 1 65 100  
i 4 11 1 67 100  
i 4 13 1 69 100  
i 4 15 3 71 100  
i 4 18 3 72 100  
e
```

## Credits

By: Andrés Cabrera and Michael Gogins

Uses code from Hermann Seib's VSTHost and Thomas Grill's vst~ object.

VST is a trademark of Steinberg Media Technologies GmbH. VST Plug-In Technology by Steinberg.

## vstnote

vstnote — Sends a MIDI note with definite duration to a VST plugin.

## Syntax

```
vstnote instance, kchan, knote, kveloc, kdur
```

## Description

*vstnote* sends a MIDI note with definite duration to a VST plugin.

## Initialization

*instance* - the number which identifies the plugin generated by *vstinit*.

## Performance

*kchan* - The MIDI channel to transmit the note on. Note that MIDI channels are numbered starting from 0.

*knote* - The MIDI note number to send.

*kveloc* - The MIDI note's velocity.

*kdur* - The MIDI note's duration in seconds.

Note: Be sure the instrument containing vstnote is not finished before the duration of the note, otherwise you'll have a 'hung' note.

## Examples

Here is an example of the use of the *vstnote* opcode. It uses the file *vst4cs.csd* [examples/vst4cs.csd].

### Example 1106. Example of the *vstnote* opcode

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Credits: Adapted by Michael Gogins
; from code by David Horowitz and Lian Cheung.
; The "--displays" option is required in order for
; the Pianoteq GUI to dispatch events and display properly.
-m3 --displays -odac
</CsOptions>
<CsInstruments>
sr      = 44100
ksmps  = 20
nchnls = 2
; Load the Pianoteq into memory.
gipianoteq vstinit "C:\\Program Files\\Steinberg\\VstPlugins\\Pianoteq 3.0 Trial\\Pianoteq
```

```

; Print information about the Pianoteq, such as parameter names and numbers.
vstinfo          gipianoteq

; Open the Pianoteq's GUI.
vstedit          gipianoteq

; Send notes from the score to the Pianoteq.
instr 1
; MIDI channels are numbered starting at 0.
; p3 always contains the duration of the note.
; p4 contains the MIDI key number (pitch),
; p5 contains the MIDI velocity number (loudness),
imidichannel    init          0
vstnote         gipianoteq, imidichannel, p4, p5, p3
endin

; Send parameter changes to the Pianoteq.
instr 2
; p4 is the parameter number.
; p5 is the parameter value.
vstparamset     gipianoteq, p4, p5
endin

; Send audio from the Pianoteq to the output.
instr 3
ablankinput     init          0
aleft, aright   vstaudio      gipianoteq, ablankinput, ablankinput
outs           left, aright
endin

</CsInstruments>
<CsScore>
; Turn on the instrument that receives audio from the Pianoteq indefinitely.
i 3 0 -1
; Send parameter changes to Pianoteq before sending any notes.
; NOTE: All parameters must be between 0.0 and 1.0.
; Length of piano strings:
i 2 0 1 33 0.5
; Hammer noise:
i 2 0 1 25 0.1
; Send a C major 7th arpeggio to the Pianoteq.
i 1 1 10 60 76
i 1 2 10 64 73
i 1 3 10 67 70
i 1 4 10 71 67
; End the performance, leaving some time
; for the Pianoteq to finish sending out its audio,
; or for the user to play with the Pianoteq virtual keyboard.
e 20
</CsScore>
</CsoundSynthesizer>

```

## Credits

By: Andrés Cabrera and Michael Gogins

Uses code from Hermann Seib's VSTHost and Thomas Grill's vst~ object.

VST is a trademark of Steinberg Media Technologies GmbH. VST Plug-In Technology by Steinberg.

# vstparamset, vstparamget

vstparamset — Used for parameter communication to and from a VST plugin.

## Syntax

```
vstparamset instance, kparam, kvalue
kvalue vstparamget instance, kparam
```

## Description

*vstparamset* and *vstparamget* are used for parameter communication to and from a VST plugin.

## Initialization

*instance* - the number which identifies the plugin, to be passed to other vst4cs opcodes.

## Performance

*kparam* - The number of the parameter to set or get.

*kvalue* - the value to set, or the the value returned by the plugin.

Parameters vary according to the plugin. To find out what parameters are available, use the verbose option when loading the plugin with vstinit. Note that the VST protocol specifies that all parameter values must lie between 0.0 and 1.0, inclusive.

## Examples

Here is an example of the use of the *vstparamset* opcode. It uses the file *vst4cs.csd* [examples/vst4cs.csd].

### Example 1107. Example of the *vstparamset* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Credits: Adapted by Michael Gogins
; from code by David Horowitz and Lian Cheung.
; The "--displays" option is required in order for
; the Pianoteq GUI to dispatch events and display properly.
-m3 --displays -odac
</CsOptions>
<CsInstruments>
sr      = 44100
ksmps  = 20
nchnls = 2

; Load the Pianoteq into memory.
gipianoteq vstinit "C:\\Program Files\\Steinberg\\VstPlugins\\Pianoteq 3.0 Trial\\Pianoteq

; Print information about the Pianoteq, such as parameter names and numbers.
vstinfo gipianoteq
```

```

; Open the Pianoteq's GUI.
vstedit          gipianoteq

; Send notes from the score to the Pianoteq.
instr 1
; MIDI channels are numbered starting at 0.
; p3 always contains the duration of the note.
; p4 contains the MIDI key number (pitch),
; p5 contains the MIDI velocity number (loudness),
imidichannel    init          0
vstnote         gipianoteq, imidichannel, p4, p5, p3
endin

; Send parameter changes to the Pianoteq.
instr 2
; p4 is the parameter number.
; p5 is the parameter value.
vstparamset     gipianoteq, p4, p5
endin

; Send audio from the Pianoteq to the output.
instr 3
ablankinput     init          0
aleft, aright   vstaudio      gipianoteq, ablankinput, ablankinput
outs          aleft, aright
endin

</CsInstruments>
<CsScore>
; Turn on the instrument that receives audio from the Pianoteq indefinitely.
i 3 0 -1
; Send parameter changes to Pianoteq before sending any notes.
; NOTE: All parameters must be between 0.0 and 1.0.
; Length of piano strings:
i 2 0 1 33 0.5
; Hammer noise:
i 2 0 1 25 0.1
; Send a C major 7th arpeggio to the Pianoteq.
i 1 1 10 60 76
i 1 2 10 64 73
i 1 3 10 67 70
i 1 4 10 71 67
; End the performance, leaving some time
; for the Pianoteq to finish sending out its audio,
; or for the user to play with the Pianoteq virtual keyboard.
e 20
</CsScore>
</CsoundSynthesizer>

```

## Credits

By: Andrés Cabrera and Michael Gogins

Uses code from Hermann Seib's VSTHost and Thomas Grill's vst~ object.

VST is a trademark of Steinberg Media Technologies GmbH. VST Plug-In Technology by Steinberg.

# vstprogset

vstprogset — Loads parameter banks to a VST plugin.

## Syntax

```
vstprogset instance, kprogram
```

## Description

*vstprogset* sets one of the programs in an *.fxb* bank.

## Initialization

*instance* - the number which identifies the plugin, to be passed to other vst4cs opcodes.

*kprogram* - the number of the program to set.

## Examples

### Example 1108. Usage of vstprogset

```
/* orc */
sr      = 44100
kr      = 4410
ksmps  = 10
nchnls = 2

gihandle1 vstinit "c:/vstplugins/cheeze/cheeze machine.dll", 1

instr 4
  vstbankload gihandle1, "c:/vstplugins/cheeze/chengo'scheese.fxb"
  vstprogset  gihandle1, 4
  vstinfo     gihandle1
endin

/* sco */
i 3 0 21
i 4 1 1 57 32
e
```

## Credits

By: Andrés Cabrera and Michael Gogins

Uses code from Hermann Seib's VSTHost and Thomas Grill's vst~ object.

VST is a trademark of Steinberg Media Technologies GmbH. VST Plug-In Technology by Steinberg.

# vsubv

vsubv — Performs subtraction between two vectorial control signals

## Description

Performs subtraction between two vectorial control signals

## Syntax

```
vsubv ifn1, ifn2, kelements [, kdstoffset] [, ksrcoffset] [,kverbose]
```

## Initialization

*ifn1* - number of the table hosting the first vector to be processed

*ifn2* - number of the table hosting the second vector to be processed

## Performance

*kelements* - number of elements of the two vectors

*kdstoffset* - index offset for the destination (*ifn1*) table (Default=0)

*ksrcoffset* - index offset for the source (*ifn2*) table (Default=0)

*kverbose* - Selects whether or not warnings are printed (Default=0)

*vsubv* subtracts two vectorial control signals, that is, each element of *ifn2* is subtracted from the corresponding element of *ifn1*. Each vectorial signal is hosted by a table (*ifn1* and *ifn2*). The number of elements contained in both vectors must be the same.

The result is a new vectorial control signal that overrides old values of *ifn1*. If you want to keep the old *ifn1* vector, use *vcopy\_i* opcode to copy it in another table. You can use *kdstoffset* and *ksrcoffset* to specify vectors in any location of the tables.

Negative values for *kdstoffset* and *ksrcoffset* are acceptable. If *kdstoffset* is negative, the out of range section of the vector will be discarded. If *ksrcoffset* is negative, the out of range elements will be assumed to be 0 (i.e. the destination elements will not be changed). If elements for the destination vector are beyond the size of the table (including guard point), these elements are discarded (i.e. elements do not wrap around the tables). If elements for the source vector are beyond the table length, these elements are taken as 0 (i.e. the destination vector will not be changed for these elements).

If the optional *kverbose* argument is different to 0, the opcode will print warning messages every k-pass if table lengths are exceeded.



### Warning

Using the same table as source and destination table in versions earlier than 5.04, might produce unexpected behavior, so use with care.

This opcode works at k-rate (this means that every k-pass the vectors are subtracted). There's an i-rate version of this opcode called *vsubv\_i*.



## Note

Please note that the *elements* argument has changed in version 5.03 from i-rate to k-rate. This will change the opcode's behavior in the unusual cases where the i-rate variable *ielements* is changed inside the instrument, for example in:

```

instr 1
ielements =      10
      vadd      1, 1, ielements
ielements =      20
      vadd      2, 1, ielements
      turnoff
endin

```

All these operators (*vaddv*, *vsubv*, *vmultv*, *vdivv*, *vpowv*, *vexpv*, *vcopy* and *vmap*) are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2*, etc.

## Examples

Here is an example of the *vsubv* opcode. It uses the file *vsubv.csd* [examples/vsubv.csd].

### Example 1109. Example of the *vsubv* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o cigoto.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr=44100
ksmps=128
nchnls=2

instr 1
ifn1 = p4
ifn2 = p5
ielements = p6
idstoffset = p7
isrcoffset = p8
kval init 25
vsubv ifn1, ifn2, ielements, idstoffset, isrcoffset, 1
endin

instr 2 ;Printtable
itable = p4
isize = ftlen(itable)
kcount init 0
kval table kcount, itable
printk2 kval

if (kcount == isize) then
  turnoff
endif

```

```
kcount = kcount + 1
  endin

</CsInstruments>
<CsScore>

f 1 0 16 -7 1 15 16

f 2 0 16 -7 1 15 2

i2 0.0 0.2 1
i2 0.2 0.2 2
i1 0.4 0.01 1 2 5 3 8
i2 0.8 0.2 1
i1 1.0 0.01 1 2 5 10 -2
i2 1.2 0.2 1
i1 1.4 0.01 1 2 8 14 0
i2 1.6 0.2 1
i1 1.8 0.01 1 2 8 0 14
i2 2.0 0.2 1
i1 2.2 0.002 1 1 8 5 2
i2 2.4 0.2 1
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado. Optional arguments added by Andres Cabrera and Istvan Varga.

New in Csound 5 (Previously available only on CsoundAV)

# vsubv\_i

`vsubv_i` — Performs subtraction between two vectorial control signals at init time.

## Description

Performs subtraction between two vectorial control signals at init time.

## Syntax

```
vsubv_i ifn1, ifn2, ielements [, idstoffset] [, isrcoffset]
```

## Initialization

*ifn1* - number of the table hosting the first vector to be processed

*ifn2* - number of the table hosting the second vector to be processed

*ielements* - number of elements of the two vectors

*idstoffset* - index offset for the destination (*ifn1*) table (Default=0)

*isrcoffset* - index offset for the source (*ifn2*) table (Default=0)

## Performance

`vsubv_i` subtracts two vectorial control signals, that is, each element of *ifn2* is subtracted from the corresponding element of *ifn1*. Each vectorial signal is hosted by a table (*ifn1* and *ifn2*). The number of elements contained in both vectors must be the same.

The result is a new vectorial control signal that overrides old values of *ifn1*. If you want to keep the old *ifn1* vector, use `vcopy_i` opcode to copy it in another table. You can use *idstoffset* and *isrcoffset* to specify vectors in any location of the tables.

Negative values for *idstoffset* and *isrcoffset* are acceptable. If *idstoffset* is negative, the out of range section of the vector will be discarded. If *isrcoffset* is negative, the out of range elements will be assumed to be 0 (i.e. the destination elements will not be changed). If elements for the destination vector are beyond the size of the table (including guard point), these elements are discarded (i.e. elements do not wrap around the tables). If elements for the source vector are beyond the table length, these elements are taken as 0 (i.e. the destination vector will not be changed for these elements).



### Warning

Using the same table as source and destination table in versions earlier than 5.04, might produce unexpected behavior, so use with care.

This opcode works at init time. There's an k-rate version of this opcode called `vsubv`.

All these operators (`vaddv_i`, `vsubv_i`, `vmultv_i`, `vdivv_i`, `vpowv_i`, `vexpv_i`, `vcopy` and `vmap`) are designed to be used together with other opcodes that operate with vectorial signals such as `vcella`, `adsynt`, `adsynt2` etc.

## Credits

Written by Gabriel Maldonado. Optional arguments added by Andres Cabrera and Istvan Varga.

New in Csound 5 (Previously available only on CsoundAV)

# vtable1k

vtable1k — Read a vector (several scalars simultaneously) from a table.

## Description

This opcode reads vectors from tables at k-rate.

## Syntax

```
vtable1k kfn,kout1 [, kout2, kout3, .... , koutN ]
```

## Performance

*kfn* - table number

*kout1...koutN* - output vector elements

*vtable1k* is a reduced version of *vtablek*, it only allows to access the first vector (it is equivalent to *vtablek* with *kndx* = zero, but a bit faster). It is useful to easily and quickly convert a set of values stored in a table into a set of k-rate variables to be used in normal opcodes, instead of using individual *table* opcodes for each value.



### Note

*vtable1k* is an unusual opcode as it produces its output on the right side arguments of the opcode.

## Examples

Here is an example of the *vtable1k* opcode. It uses the files *vtable1k.csd* [examples/vtable1k.csd].

### Example 1110. Example of the vtable1k opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d      ;;RT audio I/O
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 100
nchnls = 2

giElem init 13
giOutTab ftgen 1,0,128, 2, 0
giFreqTab ftgen 2,0,128,-7, 1,giElem, giElem+1
giSine ftgen 3,0,256,10, 1

FLpanel "This Panel contains a Slider Bank",500,400
FLslidBnk "mod1@mod2@mod3@amp@freq1@freq2@freq3@freqPo", giElem, giOutTab, 360, 600, 100, 10
```

```
FLpanel_end

FLrun

instr 1

kout1 init 0
kout2 init 0
kout3 init 0
kout4 init 0
kout5 init 0
kout6 init 0
kout7 init 0
kout8 init 0

db(kout1 * 80 )
kmodindex2= 2 * db(kout2 * 80 )
kmodindex3= 2 * db(kout3 * 80 )
kamp = 50 * db(kout4 * 70 )
kfreq1 = 1.1 * octave(kout5 * 10)
kfreq2 = 1.1 * octave(kout6 * 10)
kfreq3 = 1.1 * octave(kout7 * 10)
kfreq4 = 30 * octave(kout8 * 8)

amod1 oscili kmodindex1, kfreq1, giSine
amod2 oscili kmodindex2, kfreq2, giSine
amod3 oscili kmodindex3, kfreq3, giSine
aout oscili kamp,          kfreq4+amod1+amod2+amod3, giSine

outs aout, aout
endin

</CsInstruments>
<CsScore>

i1 0 3600
f0 3600

</CsScore>
</CsoundSynthesizer>
```

## See Also

*vtablek*

## Credits

Written by Gabriel Maldonado.

New in Csound 5.06

# vtablei

vtablei — Read vectors (from tables -or arrays of vectors).

## Description

This opcode reads vectors from tables.

## Syntax

```
vtablei indx, ifn, interp, ixmode, iout1 [ , iout2, iout3, .... , ioutN ]
```

## Initialization

*indx* - Index into f-table, either a positive number range matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* != 0).

*ifn* - table number

*iout1...ioutN* - output vector elements

*ixmode* - index data mode. The default value is 0.

== 0 index is treated as a raw table location,

== 1 index is normalized (0 to 1).

*interp* - vtable (vector table) family of opcodes allows the user to switch between interpolated or non-interpolated output by means of the *interp* argument.

## Performance

This opcode is useful in all cases in which one needs to access sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.). The number of elements of each vector (length of the vector) is determined by the number of optional arguments on the right (*iout1* , *iout2*, *iout3*, .... *ioutN*).

*vtable* (vector table) family of opcodes allows the user to switch between interpolated or non-interpolated output by means of the *interp* argument.

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using *vtable*, in order to correct eventual out-of-range values.



### Note

Notice that *vtablei*'s output arguments are placed at the right of the opcode name, differently from usual (this style is already used in other opcodes using undefined lists of output arguments such as *fin* or *trigseq*).

## Examples

Here is an example of the *vtablei* opcode. It uses the files *vtablei.csd* [examples/vtablei.csd]

**Example 1111. Example of the vtablei opcode.**

```
<CsoundSynthesizer>
<CsOptions>
-odac -B441 -b441
</CsOptions>
<CsInstruments>

sr      =      44100
kr      =      100
ksmps   =      441
nchnls  =      2

gindx init 0

      instr 1
kindex init 0
ktrig metro 0.5
if ktrig = 0 goto noevent
event "i", 2, 0, 0.5, kindex
kindex = kindex + 1
noevent:

      endin

      instr 2
iout1 init 0
iout2 init 0
iout3 init 0
iout4 init 0
indx = p4
vtablei indx, 1, 1, 0, iout1,iout2, iout3, iout4
print iout1, iout2, iout3, iout4
turnoff
      endin

</CsInstruments>
<CsScore>
f 1 0 32 10 1
i 1 0 20

</CsScore>
</CsoundSynthesizer>
```

## See also

*vtablea*, *vtablek*, *vtabi*, *vtablewi*, *vtabwi*,

## Credits

Example written by Andres Cabrera.

New in Csound 5 (Previously available only on CsoundAV)

# vtablek

vtablek — Read vectors (from tables -or arrays of vectors).

## Description

This opcode reads vectors from tables at k-rate.

## Syntax

```
vtablek kndx, kfn, kinterp, ixmode, kout1 [, kout2, kout3, .... , koutN ]
```

## Initialization

*ixmode* - index data mode. The default value is 0.

== 0 index is treated as a raw table location,

== 1 index is normalized (0 to 1).

*kinterp* - switch between interpolated or non-interpolated output. 0 -> non-interpolation , non-zero -> interpolation activated

## Performance

*kndx* - Index into f-table, either a positive number range matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* != 0).

*kfn* - table number

*kout1...koutN* - output vector elements

This opcode is useful in all cases in which one needs to access sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.). The number of elements of each vector (length of the vector) is determined by the number of optional arguments on the right (*kout1* , *kout2*, *kout3*, .... *koutN*).

*vtablek* allows the user to switch between interpolated or non-interpolated output at k-rate by means of *kinterp* argument.

*vtablek* allows also to switch the table number at k-rate (but this is possible only when vector frames of each used table have the same number of elements, otherwise unpredictable results could occur), as well as to choose indexing style (raw or normalized, see also *ixmode* argument of table opcode ).

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using *vtable*, in order to correct eventual out-of-range values.



### Note

Notice that *vtablek*'s output arguments are placed at the left of the opcode name, differently from usual (this style is already used in other opcodes using undefined lists of output arguments such as *fin* or *trigseq*).

## Examples

Here is an example of the `vtablek` opcode. It uses the files `vtablek.csd` [examples/vtablek.csd].

### Example 1112. Example of the `vtablek` opcode.

```
<CsoundSynthesizer>
<CsOptions>
-odac -B441 -b441
</CsOptions>
<CsInstruments>

sr      =      44100
kr      =      100
ksmps   =      441
nchnls  =      2

gkindx init -1

      instr 1
kindex init 0
ktrig metro 0.5
if ktrig = 0 goto noevent
gkindx = gkindx + 1
noevent:

      endin

      instr 2
kout1 init 0
kout2 init 0
kout3 init 0
kout4 init 0
vtablek gkindx, 1, 1, 0, kout1,kout2, kout3, kout4
printk2 kout1
printk2 kout2
printk2 kout3
printk2 kout4
      endin

</CsInstruments>
<CsScore>
f 1 0 32 10 1
i 1 0 20
i 2 0 20
</CsScore>
</CsoundSynthesizer>
```

## See also

*vtablea*, *vtablei*, *vtabk*, *vtablewk*, *vtabwk*,

## Credits

Written by Gabriel Maldonado.

Example written by Andres Cabrera.

New in Csound 5 (Previously available only on CsoundAV)

# vtablea

vtablea — Read vectors (from tables -or arrays of vectors).

## Description

This opcode reads vectors from tables at a-rate.

## Syntax

```
vtablea andx, kfn, kinterp, ixmode, aout1 [, aout2, aout3, .... , aoutN ]
```

## Initialization

*ixmode* - index data mode. The default value is 0.

== 0 index is treated as a raw table location,

== 1 index is normalized (0 to 1).

## Performance

*andx* - Index into f-table, either a positive number range matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* != 0).

*kfn* - table number

*kinterp* - switch between interpolated or non-interpolated output. 0 -> non-interpolation , non-zero -> interpolation activated

*aout1...aoutN* - output vector elements

This opcode is useful in all cases in which one needs to access sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.). The number of elements of each vector (length of the vector) is determined by the number of optional arguments on the right (*aout1* , *aout2*, *aout3*, .... *aoutN*).

**vtablea** allows the user to switch between interpolated or non-interpolated output at k-rate by means of *kinterp* argument.

**vtablea** allows also to switch the table number at k-rate (but this is possible only when vector frames of each used table have the same number of elements, otherwise unpredictable results could occur), as well as to choose indexing style (raw or normalized, see also *ixmode* argument of table opcode ).

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using *vtablea*, in order to correct eventual out-of-range values.



### Note

Notice that *vtablea*'s output arguments are placed at the right of the opcode name, differently from usual (this style is already used in other opcodes using undefined lists of output arguments such as *fin* or *trigseq*).

## See also

*vtablek, vtablei, vtaba, vtablewa, vtabwa,*

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vtablewi

vtablewi — Write vectors (to tables -or arrays of vectors).

## Description

This opcode writes vectors to tables at init time.

## Syntax

```
vtablewi indx, ifn, ixmode, inarg1 [ , inarg2, inarg3 , ... , inargN ]
```

## Initialization

*indx* - Index into f-table, either a positive number range matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* != 0).

*ifn* - table number

*ixmode* - index data mode. The default value is 0.

== 0 index is treated as a raw table location,

== 1 index is normalized (0 to 1).

*inarg1...inargN* - input vector elements

## Performance

This opcode is useful in all cases in which one needs to write sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.). The number of elements of each vector (length of the vector) is determined by the number of optional arguments on the right (*inarg1, inarg2, inarg3, .... inargN*).

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using *vtablewi*, in order to correct eventual out-of-range values.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vtablewk

vtablewk — Write vectors (to tables -or arrays of vectors).

## Description

This opcode writes vectors to tables at k-rate.

## Syntax

```
vtablewk kndx, kfn, ixmode, kinarg1 [ , kinarg2, kinarg3 , .... , kinargN ]
```

## Initialization

*ixmode* - index data mode. The default value is 0.

== 0 index is treated as a raw table location,

== 1 index is normalized (0 to 1).

## Performance

*kndx* - Index into f-table, either a positive number range matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* != 0).

*kfn* - table number

*kinarg1...kinargN* - input vector elements

This opcode is useful in all cases in which one needs to write sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.). The number of elements of each vector (length of the vector) is determined by the number of optional arguments on the right (*kinarg1*, *kinarg2*, *kinarg3*, .... *kinargN*).

**vtablewk** allows also to switch the table number at k-rate (but this is possible only when vector frames of each used table have the same number of elements, otherwise unpredictable results could occur), as well as to choose indexing style (raw or normalized, see also *ixmode* argument of *table* opcode ).

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using *vtablewk*, in order to correct eventual out-of-range values.

## Examples

Here is an example of the vtablewk opcode. It uses the files *vtablewk.csd* [examples/vtablewk.csd].

### Example 1113. Example of the vtablewk opcode.

```
<CsoundSynthesizer>  
<CsOptions>
```

```
;-ovtablewa.wav -W -b441 -B441
-odac -b441 -B441
</CsOptions>
<CsInstruments>

  sr=44100
  kr=441
  ksmps=100
  nchnls=2

  instr 1
  ilen = ftlen(1)

  knew1 oscil 10000, 440, 3
  knew2 oscil 15000, 440, 3, 0.5
  kindex phasor 0.3
  asig oscil 1, sr/ilen, 1
  vtablewk kindex*ilen, 1, 0, knew1, knew2
  out asig,asig
  endin

</CsInstruments>
<CsScore>
f1 0 262144 -1 "beats.wav" 0 4 0
f2 0 262144 2 0
f3 0 1024 10 1

i1 0 10
</CsScore>
</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado.

Example written by Andres Cabrera.

New in Csound 5 (Previously available only on CsoundAV)

# vtablewa

vtablewa — Write vectors (to tables -or arrays of vectors).

## Description

This opcode writes vectors to tables at a-rate.

## Syntax

```
vtablewa andx, kfn, ixmode, ainarg1 [ , ainarg2, ainarg3 , .... , ainargN ]
```

## Initialization

*ixmode* - index data mode. The default value is 0.

== 0 index is treated as a raw table location,

== 1 index is normalized (0 to 1).

## Performance

*andx* - Index into f-table, either a positive number range matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* != 0).

*kfn* - table number

*ainarg1...ainargN* - input vector elements

This opcode is useful in all cases in which one needs to write sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.). The number of elements of each vector (length of the vector) is determined by the number of optional arguments on the right (*ainarg1* , *ainarg2*, *ainarg3*, .... *ainargN*).

*vtablewa* allows also to switch the table number at k-rate (but this is possible only when vector frames of each used table have the same number of elements, otherwise unpredictable results could occur), as well as to choose indexing style (raw or normalized, see also *ixmode* argument of table opcode ).

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using *vtablewa*, in order to correct eventual out-of-range values.

## Examples

Here is an example of the *vtablewa* opcode. It uses the files *vtablewa.csd* [examples/vtablewa.csd].

### Example 1114. Example of the vtablewa opcode.

```
<CsoundSynthesizer>  
<CsOptions>
```

```
-odac -b441 -B441
</CsOptions>
<CsInstruments>

  sr=44100
  kr=4410
  ksmpr=10
  nchnls=2
  odbfs = 1

  instr 1
  vcopy 2, 1, 262144
  ar random 0, 1
  vtablewa ar,2,1,ar
  out ar,ar
  endin

</CsInstruments>
<CsScore>
f1 0 262144 -1 "beats.wav" 0 4 0
f2 0 262144 2 0

i1 0 4

</CsScore>
</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado.

Example written by Andres Cabrera.

New in Csound 5 (Previously available only on CsoundAV)

# vtabi

vtabi — Read vectors (from tables -or arrays of vectors).

## Description

This opcode reads vectors from tables.

## Syntax

```
vtabi indx, ifn, iout1 [, iout2, iout3, .... , ioutN ]
```

## Initialization

*indx* - Index into f-table, either a positive number range matching the table length

*ifn* - table number

*iout1...ioutN* - output vector elements

## Performance

This opcode is useful in all cases in which one needs to access sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.). The number of elements of each vector (length of the vector) is determined by the number of optional arguments on the right (*iout1*, *iout2*, *iout3*, .... *ioutN*).

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using *vtabi*, in order to correct eventual out-of-range values.

The **vtab** family is similar to **vtable**, but is much faster because interpolation is not available, table number cannot be changed after initialization, and only raw indexing is supported.



### Note

Notice that *vtabi*'s output arguments are placed at the right of the opcode name, differently from usual (this style is already used in other opcodes using undefined lists of output arguments such as *fin* or *trigseq*).

## Examples

For an example of the *vtabi* opcode usage, see *vtablei*.

## See also

*vtabk*, *vtaba*, *vtablei*, *vtablewi*, *vtabwi*,

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vtabk

vtabk — Read vectors (from tables -or arrays of vectors).

## Description

This opcode reads vectors from tables at k-rate.

## Syntax

```
vtabk kndx, ifn, kout1 [ , kout2, kout3, .... , koutN ]
```

## Initialization

*ifn* - table number

## Performance

*kndx* - Index into f-table, either a positive number range matching the table length

*kout1...koutN* - output vector elements

This opcode is useful in all cases in which one needs to access sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.) . The number of elements of each vector (length of the vector) is determined by the number of optional arguments on the right (*kout1*, *kout2*, *kout3*, .... *koutN*).

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using *vtabk*, in order to correct eventual out-of-range values.

The **vtab** family is similar to **vtable**, but is much faster because interpolation is not available, table number cannot be changed after initialization, and only raw indexing is supported.



### Note

Notice that *vtabk*'s output arguments are placed at the right of the opcode name, differently from usual (this style is already used in other opcodes using undefined lists of output arguments such as *fin* or *trigseq*).

## Examples

For an example of the vtabk opcode usage, see *vtablek*.

## See also

*vtabi*, *vtaba*, *vtablek*, *vtablewk*, *vtabwk*,

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vtaba

vtaba — Read vectors (from tables -or arrays of vectors).

## Description

This opcode reads vectors from tables at a-rate.

## Syntax

```
vtaba andx, ifn, aout1 [ , aout2, aout3, .... , aoutN ]
```

## Initialization

*ifn* - table number

## Performance

*andx* - Index into f-table, either a positive number range matching the table length

*aout1...aoutN* - output vector elements

This opcode is useful in all cases in which one needs to access sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.). The number of elements of each vector (length of the vector) is determined by the number of optional arguments on the right (*aout1*, *aout2*, *aout3*, .... *aoutN*).

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using *vtaba*, in order to correct eventual out-of-range values.

The **vtab** family is similar to the **vtable** family, but is much faster because interpolation is not available, table number cannot be changed after initialization, and only raw indexing is supported.



### Note

Notice that *vtaba*'s output arguments are placed at the right of the opcode name, differently from usual (this style is already used in other opcodes using undefined lists of output arguments such as *fin* or *trigseq*).

## Examples

The usage of *vtaba* is similar to *vtablek*.

## See also

*vtabk*, *vtabi*, *vtablea*, *vtablewa*, *vtabwa*,

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vtabwi

vtabwi — Write vectors (to tables -or arrays of vectors).

## Description

This opcode writes vectors to tables at init time.

## Syntax

```
vtabwi indx, ifn, inarg1 [, inarg2, inarg3 , ... , inargN ]
```

## Initialization

*indx* - Index into f-table, a positive number range matching the table length

*ifn* - table number

*inarg1...inargN* - input vector elements

## Performance

This opcode is useful in all cases in which one needs to write sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.). The number of elements of each vector (length of the vector) is determined by the number of optional arguments on the right (*inarg1, inarg2, inarg3, .... inargN*).

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using *vtabwi*, in order to correct eventual out-of-range values.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vtabwk

vtabwk — Write vectors (to tables -or arrays of vectors).

## Description

This opcode writes vectors to tables at k-rate.

## Syntax

```
vtabwk kndx, ifn, kinarg1 [ , kinarg2, kinarg3 , ... , kinargN ]
```

## Initialization

*ifn* - table number

## Performance

*kndx* - Index into f-table, a positive number range matching the table length

*kinarg1...kinargN* - input vector elements

This opcode is useful in all cases in which one needs to write sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.) . The number of elements of each vector (length of the vector) is determined by the number of optional arguments on the right (*kinarg1*, *kinarg2*, *kinarg3*, .... *kinargN*).

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using *vtabwk*, in order to correct eventual out-of-range values.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vtabwa

vtabwa — Write vectors (to tables -or arrays of vectors).

## Description

This opcode writes vectors to tables at a-rate.

## Syntax

```
vtabwa andx, ifn, ainarg1 [ , ainarg2, ainarg3 , .... , ainargN ]
```

## Initialization

*ifn* - table number

## Performance

*andx* - Index into f-table, a positive number range matching the table length

*ainarg1...ainargN* - input vector elements

This opcode is useful in all cases in which one needs to write sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.). The number of elements of each vector (length of the vector) is determined by the number of optional arguments on the right (*ainarg1*, *ainarg2*, *ainarg3*, .... *ainargN*).

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using *vtabwa*, in order to correct eventual out-of-range values.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vwrap

vwrap — Limiting and Wrapping Vectorial Signals

## Description

Wraps elements of vectorial control signals.

## Syntax

```
vwrap ifn, kmin, kmax, ielements
```

## Initialization

*ifn* - number of the table hosting the vector to be processed

*ielements* - number of elements of the vector

## Performance

*kmin* - minimum threshold value

*kmax* - maximum threshold value

*vwrap* wraps around each element of corresponding vector if it exceeds low or high thresholds.

These opcodes are similar to *limit*, *wrap* and *mirror*, but operate with a vectorial signal instead of with a scalar signal.

Result overrides old values of *ifn1*, if these are out of min/max interval. If you want to keep input vector, use *vcopy* opcode to copy it in another table.

All these opcodes are designed to be used together with other opcodes that operate with vectorial signals such as *vcella*, *adsynt*, *adsynt2* etc.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# waveset

waveset — A simple time stretch by repeating cycles.

## Description

A simple time stretch by repeating cycles.

## Syntax

```
ares waveset ain, krep [, ilen]
```

## Initialization

*ilen* (optional, default=0) -- the length (in samples) of the audio signal. If *ilen* is set to 0, it defaults to half the given note length (p3).

## Performance

*ain* -- the input audio signal.

*krep* -- the number of times the cycle is repeated.

The input is read and each complete cycle (two zero-crossings) is repeated *krep* times.

There is an internal buffer as the output is clearly slower than the input. Some care is taken if the buffer is too short, but there may be strange effects.

## Examples

Here is an example of the waveset opcode. It uses the file *waveset.csd* [examples/waveset.csd].

### Example 1115. Example of the waveset opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o waveset.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs  =1

instr 1
```

```
krep init p4
asig soundin "flute.aiff"
aout waveset asig, krep
      outs aout, aout

endin
</CsInstruments>
<CsScore>

i 1 0 3 1 ;no repetitions
i 1 + 10 3 ;stretching 3 times
i 1 + 14 6 ;6 times

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitth  
February 2001

New in version 4.11

# websocket

websocket — Read and write signals and arrays using a websocket connection.

## Description

*websocket* reads and writes  $N$  signals and arrays using a websocket connection.

## Syntax

```
xout1[, xout2, xout3, ..., xoutN] websocket iport, xin
```

## Initialization

*iport* the local web port to read/write data.

## Performance

## Performance

*xout1, ... xoutN* -- The output variables which contain data received from a websocket. On the web side the websocket must send data using a protocol name that matches the output variable name e.g. "ksignal" for a k-rate variable. If an array is intended to be received from a websocket it must be first initialised before being used as an output to the opcode. Otherwise the opcode doesn't know what size data to expect from the websocket. When sending data to a websocket from the web page it must be sent as a 32 or 64 bit array, depending on the build of Csound that is being used.

*xin* -- The input variable which contains data to be sent to a websocket. On the web side the websocket receives data using a protocol name that matches the input variable name e.g. "ksignal" for a k-rate variable. When receiving data from a websocket on the web page it must be read as a 32 or 64 bit array, depending on the build of Csound that is being used.



### Note

The total number of input and output arguments is limited to 20.

A-rate variables must be send and received as arrays that are *ksmps* samples large. A-rate arrays similarly are sent and received as *ksmps* by the number of elements in the array. K-rate variables are sent and received as a single element array. K-rate arrays are sent and received as arrays with matching numbers of elements.

## Examples

Here is a simple example of the websocket opcode. It uses the file *websocket.csd* [examples/websocket.csd] and *websocket.html* [examples/websocket.html].

### Example 1116. Example of the websocket opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
nchnls = 2
0dbfs = 1
ksmps = 256
sr = 44100

schedule 1, 0, -1

instr 1
  klfo lfo 1, 1
  iport init 8888
  kinput websocket iport, klfo
  printk2 kinput
endin

</CsInstruments>
<CsScore>
</CsScore>
</CsoundSynthesizer>
```

## See Also

*OSListen OSCsend*

## Credits

Author: Edward Costello;  
NUIM, 2015

New in version 6.06

# weibull

weibull — Weibull distribution random number generator (positive values only).

## Description

Weibull distribution random number generator (positive values only). This is an x-class noise generator

## Syntax

```
ares weibull ksigma, ktau
ires weibull ksigma, ktau
kres weibull ksigma, ktau
```

## Performance

*ksigma* -- scales the spread of the distribution.

*ktau* -- if greater than one, numbers near *ksigma* are favored. If smaller than one, small values are favored. If t equals 1, the distribution is exponential. Outputs only positive numbers.

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the weibull opcode. It uses the file *weibull.csd* [examples/weibull.csd].

### Example 1117. Example of the weibull opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o weibull.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1    ; every run time same values
```

```

ktri weibull 100, 1
  printk .2, ktri ; look
aout oscili 0.8, 440+ktri, 1 ; & listen
  outs aout, aout
endin

instr 2 ; every run time different values

  seed 0
ktri weibull 100, 1
  printk .2, ktri ; look
aout oscili 0.8, 440+ktri, 1 ; & listen
  outs aout, aout
endin

instr 3 ; every run time different values

  seed 0
ktri weibull 100, 10 ; closer to ksigma..
  printk .2, ktri ; look
aout oscili 0.8, 440+ktri, 1 ; & listen
  outs aout, aout
endin
</CsInstruments>
<CsScore>
; sine wave
f 1 0 16384 10 1

i 1 0 2
i 2 3 2
i 3 6 2
e
</CsScore>
</CsoundSynthesizer>

```

Its output should include a line like these:

```

i 1 1 time 0.00073: 168.59070
i 1 1 time 0.20027: 98.72078
i 1 1 time 0.40054: 48.57351
i 1 1 time 0.60009: 171.46941
i 1 1 time 0.80036: 50.20434
i 1 1 time 1.00063: 50.84775
i 1 1 time 1.20018: 18.16301
i 1 1 time 1.40045: 44.41001
i 1 1 time 1.60000: 0.98506
i 1 1 time 1.80027: 36.19192

```

WARNING: Seeding from current time 2444541554

```

i 2 2 time 3.00045: 20.81653
i 2 2 time 3.20000: 116.17060
i 2 2 time 3.40027: 9.23891
i 2 2 time 3.59982: 95.67111
i 2 2 time 3.80009: 296.52851
i 2 2 time 4.00036: 39.28636
i 2 2 time 4.19991: 13.54326
i 2 2 time 4.40018: 54.92388
i 2 2 time 4.59973: 268.05584
i 2 2 time 4.80000: 95.27069
i 2 2 time 5.00027: 91.62076

```

WARNING: Seeding from current time 2447542341

```

i 3 3 time 6.00091: 94.40902

```

i	3	time	6.20045:	111.10193
i	3	time	6.40073:	99.38797
i	3	time	6.60027:	98.54267
i	3	time	6.80054:	106.53899
i	3	time	7.00082:	106.30752
i	3	time	7.20036:	88.75486
i	3	time	7.40063:	106.45703
i	3	time	7.60091:	84.59854
i	3	time	7.80045:	106.76515

## See Also

*seed, betarand, bexprnd, cauchy, exprand, gauss, linrand, pcauchy, poisson, trirand, unirand*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

# wgbow

wgbow — Creates a tone similar to a bowed string.

## Description

Audio output is a tone similar to a bowed string, using a physical model developed from Perry Cook, but re-coded for Csound.

## Syntax

```
ares wgbow kamp, kfreq, kpres, krat, kvibf, kvamp \  
[, ifn] [, iminfreq]
```

## Initialization

*ifn* -- optional table of shape of vibrato, defaults to a sine table.

*iminfreq* (optional) -- lowest frequency at which the instrument will play. If it is omitted it is taken to be the same as the initial *kfreq*. If *iminfreq* is negative, initialization will be skipped.

## Performance

A note is played on a string-like instrument, with the arguments as below.

*kamp* -- amplitude of note.

*kfreq* -- frequency of note played.

*kpres* -- a parameter controlling the pressure of the bow on the string. Values should be about 3. The useful range is approximately 1 to 5.

*krat* -- the position of the bow along the string. Usual playing is about 0.127236. The suggested range is 0.025 to 0.23.

*kvibf* -- frequency of vibrato in Hertz. Suggested range is 0 to 12

*kvamp* -- amplitude of the vibrato

## Examples

Here is an example of the wgbow opcode. It uses the file *wgbow.csd* [examples/wgbow.csd].

### Example 1118. Example of the wgbow opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsoundOptions>  
; Select audio/midi flags here according to platform  
-odac      ;;;realtime audio out
```

```
;-iadc    ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o wgbow.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kpres = p4      ;pressure value
krat = p5       ;position along string
kvibf = 6.12723

kvib linseg 0, 0.5, 0, 1, 1, p3-0.5, 1    ; amplitude envelope for the vibrato.
kvamp = kvib * 0.01
asig wgbow .7, 55, kpres, krat, kvibf, kvamp, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 2048 10 1 ;sine wave

i 1 0 3 3 0.127236
i 1 + 3 5 0.127236
i 1 + 3 5 0.23

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

ifn optional in version 6.06

# wgbowedbar

wgbowedbar — A physical model of a bowed bar.

## Description

A physical model of a bowed bar, belonging to the Perry Cook family of waveguide instruments.

## Syntax

```
ares wgbowedbar kamp, kfreq, kpos, kbowpres, kgain [, iconst] [, itvel] \  
    [, ibowpos] [, ilow]
```

## Initialization

*iconst* (optional, default=0) -- an integration constant. Default is zero.

*itvel* (optional, default=0) -- either 0 or 1. When *itvel* = 0, the bow velocity follows an ADSR style trajectory. When *itvel* = 1, the value of the bow velocity decays in an exponentially.

*ibowpos* (optional, default=0) -- the position on the bow, which affects the bow velocity trajectory.

*ilow* (optional, default=0) -- lowest frequency required

## Performance

*kamp* -- amplitude of signal

*kfreq* -- frequency of signal

*kpos* -- position of the bow on the bar, in the range 0 to 1

*kbowpres* -- pressure of the bow (as in *wgbowed*)

*kgain* -- gain of filter. A value of about 0.809 is suggested.

## Examples

Here is an example of the wgbowedbar opcode. It uses the file *wgbowedbar.csd* [examples/wgbowedbar.csd].

### Example 1119. Example of the wgbowedbar opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsoundOptions>  
; Select audio/midi flags here according to platform  
-odac      ;;realtime audio out  
;-iadc     ;;uncomment -iadc if RT audio input is needed too  
; For Non-realtime ouput leave only the line below:
```

```
; -o wgbowedbar.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kp = p6
asig wgbowedbar p4, cpspch(p5), 1, kp, 0.995
outs asig, asig

endin
</CsInstruments>
<CsScore>
s
i1 0 .5 .5 7.00 .1 ;short sound
i1 + . .3 8.00 .1
i1 + . .5 9.00 .1
s
i1 0 .5 .5 7.00 1 ;longer sound
i1 + . .3 8.00 1
i1 + . .5 9.00 1

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 4.07

# wgbrass

wgbrass — Creates a tone related to a brass instrument.

## Description

Audio output is a tone related to a brass instrument, using a physical model developed from Perry Cook, but re-coded for Csound.

## Syntax

```
ares wgbrass kamp, kfreq, ktens, iatt, kvibf, kvamp \  
    [, ifn] [, iminfreq]
```

## Initialization

*iatt* -- time taken to reach full pressure

*ifn* -- optional table of shape of vibrato, defaults to a sine table.

*iminfreq* -- lowest frequency at which the instrument will play. If it is omitted it is taken to be the same as the initial *kfreq*. If *iminfreq* is negative, initialization will be skipped.

## Performance

A note is played on a brass-like instrument, with the arguments as below.

*kamp* -- Amplitude of note.

*kfreq* -- Frequency of note played.

*ktens* -- lip tension of the player. Suggested value is about 0.4

*kvibf* -- frequency of vibrato in Hertz. Suggested range is 0 to 12

*kvamp* -- amplitude of the vibrato



### NOTE

This is rather poor, and at present uncontrolled. Needs revision, and possibly more parameters.

## Examples

Here is an example of the wgbrass opcode. It uses the file *wgbrass.csd* [examples/wgbrass.csd].

### Example 1120. Example of the wgbrass opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o wgbass.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
ksmps = 10
nchnls = 1
0dbfs = 1

; Instrument #1.
instr 1
  kamp = 0.7
  kfreq = p4
  ktens = p5
  iatt = p6
  kvibf = p7
  ifn = 1

; Create an amplitude envelope for the vibrato.
kvamp line 0, p3, 0.5

a1 wgbass kamp, kfreq, ktens, iatt, kvibf, kvamp, ifn
out a1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 1024 10 1

;          freq  tens  att  vibf
i 1 0 4  440    0.4  0.1  6.137
i 1 4 4  440    0.4  0.01 0.137
i 1 8 4  880    0.4  0.1  6.137
e

</CsScore>
</CsoundSynthesizer>

```

## Credits

Author: John ffitich (after Perry Cook)  
 University of Bath, Codemist Ltd.  
 Bath, UK

New in Csound version 3.47

ifn optional in 6.06

# wgclar

wgclar — Creates a tone similar to a clarinet.

## Description

Audio output is a tone similar to a clarinet, using a physical model developed from Perry Cook, but re-coded for Csound.

## Syntax

```
ares wgclar kamp, kfreq, kstiff, \  
iatt, idetk, kngain, kvibf, kvamp [, ifn] [, iminfreq]
```

## Initialization

*iatt* -- time in seconds to reach full blowing pressure. 0.1 seems to correspond to reasonable playing. A longer time gives a definite initial wind sound.

*idetk* -- time in seconds taken to stop blowing. 0.1 is a smooth ending

*ifn* (optional) -- table of shape of vibrato, defaults yo te usual sine table.

*iminfreq* (optional) -- lowest frequency at which the instrument will play. If it is omitted it is taken to be the same as the initial *kfreq*. If *iminfreq* is negative, initialization will be skipped.

## Performance

A note is played on a clarinet-like instrument, with the arguments as below.

*kamp* -- Amplitude of note.

*kfreq* -- Frequency of note played.

*kstiff* -- a stiffness parameter for the reed. Values should be negative, and about -0.3. The useful range is approximately -0.44 to -0.18.

*kngain* -- amplitude of the noise component, about 0 to 0.5

*kvibf* -- frequency of vibrato in Hertz. Suggested range is 0 to 12

*kvamp* -- amplitude of the vibrato

## Examples

Here is an example of the wgclar opcode. It uses the file *wgclar.csd* [examples/wgclar.csd].

### Example 1121. Example of the wgclar opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o wgclar.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kfreq = 330
kstiff = -0.3
iatt = 0.1
idetk = 0.1
kngain init p4 ;vary breath
kvibf = 5.735
kvamp = 0.1

asig wgclar .9, kfreq, kstiff, iatt, idetk, kngain, kvibf, kvamp, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine wave

i 1 0 2 0.2
i 1 + 2 0.5 ;more breath
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitich (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

ifn optional in 6.06

# wgflute

wgflute — Creates a tone similar to a flute.

## Description

Audio output is a tone similar to a flute, using a physical model developed from Perry Cook, but re-coded for Csound.

## Syntax

```
ares wgflute kamp, kfreq, kjet, iatt,  
      idetk, kngain, kvibf, kvamp [, ifn] [, iminfreq] [, ijetr] [, iendrf]
```

## Initialization

*iatt* -- time in seconds to reach full blowing pressure. 0.1 seems to correspond to reasonable playing.

*idetk* -- time in seconds taken to stop blowing. 0.1 is a smooth ending

*ifn* (optional) -- table of shape of vibrato, defaults to the usual sine table.

*iminfreq* (optional) -- lowest frequency at which the instrument will play. If it is omitted it is taken to be the same as the initial *kfreq*. If *iminfreq* is negative, initialization will be skipped.

*ijetr* (optional, default=0.5) -- amount of reflection in the breath jet that powers the flute. Default value is 0.5.

*iendrf* (optional, default=0.5) -- reflection coefficient of the breath jet. Default value is 0.5. Both *ijetr* and *iendrf* are used in the calculation of the pressure differential.

## Performance

*kamp* -- Amplitude of note.

*kfreq* -- Frequency of note played. While it can be varied in performance, I have not tried it.

*kjet* -- a parameter controlling the air jet. Values should be positive, and about 0.3. The useful range is approximately 0.08 to 0.56.

*kngain* -- amplitude of the noise component, about 0 to 0.5

*kvibf* -- frequency of vibrato in Hertz. Suggested range is 0 to 12

*kvamp* -- amplitude of the vibrato

## Examples

Here is an example of the wgflute opcode. It uses the file *wgflute.csd* [examples/wgflute.csd].

### Example 1122. Example of the wgflute opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o wgflute.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kfreq = 440
kjet init p4 ;vary air jet
iatt = 0.1
idetk = 0.1
kngain = 0.15
kvibf = 5.925
kvamp = 0.05

asig wgflute .8, kfreq, kjet, iatt, idetk, kngain, kvibf, kvamp, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine wave

i 1 0 2 0.02 ;more air jet
i 1 + 2 0.32
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitich (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

ifn optional in 6.06

# wgpluck

wgpluck — A high fidelity simulation of a plucked string.

## Description

A high fidelity simulation of a plucked string, using interpolating delay-lines.

## Syntax

```
ares wgpluck icps, iamp, kpick, iplk, idamp, ifilt, axcite
```

## Initialization

*icps* -- frequency of plucked string

*iamp* -- amplitude of string pluck

*iplk* -- point along the string, where it is plucked, in the range of 0 to 1. 0 = no pluck

*idamp* -- damping of the note. This controls the overall decay of the string. The greater the value of *idamp*, the faster the decay. Negative values will cause an increase in output over time.

*ifilt* -- control the attenuation of the filter at the bridge. Higher values cause the higher harmonics to decay faster.

## Performance

*kpick* -- proportion of the way along the point to sample the output.

*axcite* -- a signal which excites the string.

A string of frequency *icps* is plucked with amplitude *iamp* at point *iplk*. The decay of the virtual string is controlled by *idamp* and *ifilt* which simulate the bridge. The oscillation is sampled at the point *kpick*, and excited by the signal *axcite*.

## Examples

The following example produces a moderately long note with rapidly decaying upper partials. It uses the file *wgpluck.csd* [examples/wgpluck.csd].

### Example 1123. An example of the wgpluck opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsoundOptions>  
; Select audio/midi flags here according to platform  
; Audio out  Audio in  No messages  
-odac          -iadc     -d          ;;RT audio I/O  
; For Non-realtime ouput leave only the line below:
```

```

; -o wGPLuck.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  icps = 220
  iamp = 20000
  kpick = 0.5
  iplk = 0
  idamp = 10
  ifilt = 1000

  excite oscil 1, 1, 1
  apluck wGPLuck icps, iamp, kpick, iplk, idamp, ifilt, excite

  out apluck
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e

</CsScore>
</CsoundSynthesizer>

```

The following example produces a shorter, brighter note. It uses the file *wGPLuck\_brighter.csd* [examples/wGPLuck\_brighter.csd].

### Example 1124. An example of the wGPLuck opcode with a shorter, brighter note.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d          ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o wGPLuck_brighter.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  icps = 220
  iamp = 20000

```

```
kpick = 0.5
iplk = 0
idamp = 30
ifilt = 10

axcite oscil 1, 1, 1
apluck wgpluck icps, iamp, kpick, iplk, idamp, ifilt, axcite

out apluck
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Michael A. Casey  
M.I.T.  
Cambridge, Mass.  
1997

New in Version 3.47

# wgpluck2

wgpluck2 — Physical model of the plucked string.

## Description

*wgpluck2* is an implementation of the physical model of the plucked string, with control over the pluck point, the pickup point and the filter. Based on the Karplus-Strong algorithm.

## Syntax

```
ares wgpluck2 iplk, kamp, icps, kpick, krefl
```

## Initialization

*iplk* -- The point of pluck is *iplk*, which is a fraction of the way up the string (0 to 1). A pluck point of zero means no initial pluck.

*icps* -- The string plays at *icps* pitch.

## Performance

*kamp* -- Amplitude of note.

*kpick* -- Proportion of the way along the string to sample the output.

*krefl* -- the coefficient of reflection, indicating the lossiness and the rate of decay. It must be strictly between 0 and 1 (it will complain about both 0 and 1).

## Examples

Here is an example of the *wgpluck2* opcode. It uses the file *wgpluck2.csd* [examples/wgpluck2.csd].

### Example 1125. Example of the *wgpluck2* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o wgpluck2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1
```

```
; Instrument #1.
instr 1
  iplk = 0.75
  kamp = 30000
  icps = 220
  kpick = 0.75
  krefl = 0.5

  apluck wgpluck2 iplk, kamp, icps, kpick, krefl

  out apluck
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for two seconds.
i 1 0 2
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*repluck*

## Credits

Author: John ffitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

# wguide1

wguide1 — A simple waveguide model consisting of one delay-line and one first-order lowpass filter.

## Description

A simple waveguide model consisting of one delay-line and one first-order lowpass filter.

## Syntax

```
ares wguide1 asig, xfreq, kcutoff, kfeedback
```

## Performance

*asig* -- the input of excitation noise.

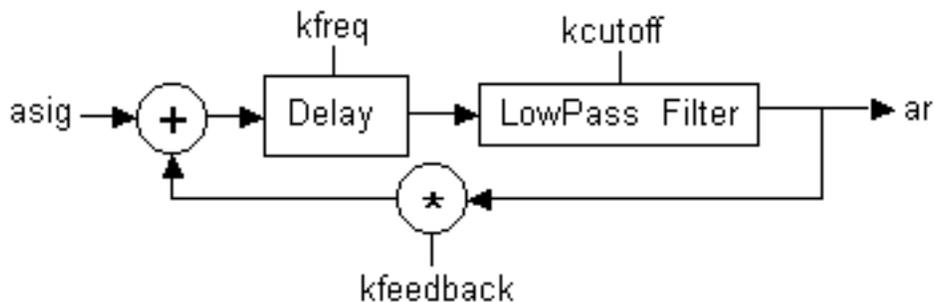
*xfreq* -- the frequency (i.e. the inverse of delay time) Changed to x-rate in Csound version 3.59.

*kcutoff* -- the filter cutoff frequency in Hz.

*kfeedback* -- the feedback factor.

*wguide1* is the most elemental waveguide model, consisting of one delay-line and one first-order lowpass filter.

Implementing waveguide algorithms as opcodes, instead of orc instruments, allows the user to set *kr* different than *sr*, allowing better performance particularly when using real-time.



wguide1.

## Examples

Here is an example of the wguide1 opcode. It uses the file *wguide1.csd* [examples/wguide1.csd].

### Example 1126. Example of the wguide1 opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc    -d      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o wguide1.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - a simple noise waveform.
instr 1
; Generate some noise.
asig noise 20000, 0.5

out asig
endin

; Instrument #2 - a waveguide example.
instr 2
; Generate some noise.
asig noise 20000, 0.5

; Run it through a wave-guide model.
kfreq init 200
kcutoff init 3000
kfeedback init 0.8
awg1 wguide1 asig, kfreq, kcutoff, kfeedback

out awg1
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for 2 seconds.
i 1 0 2
; Play Instrument #2 for 2 seconds.
i 2 2 2
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*wguide2*

## Credits

Author: Gabriel Maldonado  
Italy  
October 1998

Example written by Kevin Conder.

New in Csound version 3.49

## wguide2

`wguide2` — A model of beaten plate consisting of two parallel delay-lines and two first-order lowpass filters.

### Description

A model of beaten plate consisting of two parallel delay-lines and two first-order lowpass filters.

### Syntax

```
ares wguide2 asig, xfreq1, xfreq2, kcutoff1, kcutoff2, \  
      kfeedback1, kfeedback2
```

### Performance

*asig* -- the input of excitation noise

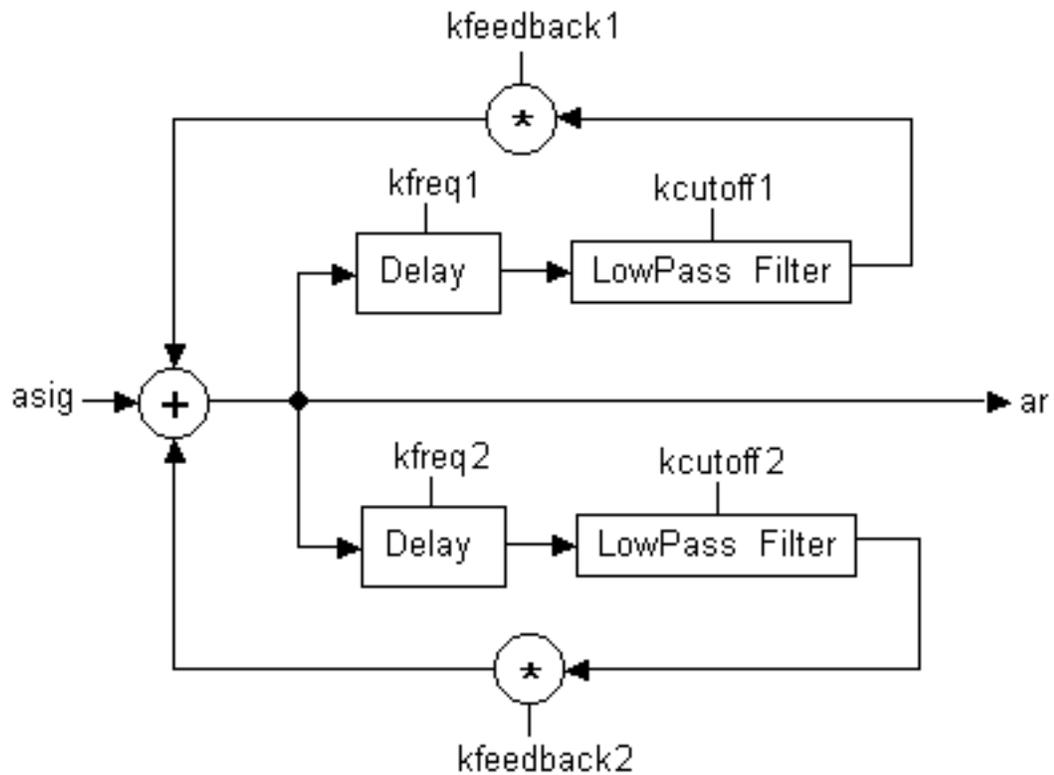
*xfreq1*, *xfreq2* -- the frequency (i.e. the inverse of delay time) Changed to x-rate in Csound version 3.59.

*kcutoff1*, *kcutoff2* -- the filter cutoff frequency in Hz.

*kfeedback1*, *kfeedback2* -- the feedback factor

`wguide2` is a model of beaten plate consisting of two parallel delay-lines and two first-order lowpass filters. The two feedback lines are mixed and sent to the delay again each cycle.

Implementing waveguide algorithms as opcodes, instead of orc instruments, allows the user to set *kr* different than *sr*, allowing better performance particularly when using real-time.



wguide2.



### Note

As a rule of thumb, to avoid making *wguide2* unstable, the sum of the two feedback values should be below 0.5.

## Examples

Here is an example of the *wguide2* opcode. It uses the file *wguide2.csd* [examples/wguide2.csd].

### Example 1127. Example of the *wguide2* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o wguide2.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
```

```
nchnls = 2
0dbfs = 1

instr 1

aout diskin2 "beats.wav", 1, 0, 1 ;in signal
afreq1 line 100, p3, 2000
afreq2 line 1200, p3, p4 ;vary second frequency in the score
kutoff1 = 3000
kutoff2 = 1500
kfeedback1 = 0.25 ;the sum of the two feedback
kfeedback2 = 0.25 ;values should not exceed 0.5
asig wguide2 aout, afreq1, afreq2, kutoff1, kutoff2, kfeedback1, kfeedback2
asig dcblock2 asig ;get rid of DC
outs asig, asig

endin
</CsInstruments>
<CsScore>
i 1 0 8 1200 ;frequency of afreq2 remains the same
i 1 9 8 100 ;frequency of afreq2 gets lower
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*wguide1*

## Credits

Author: Gabriel Maldonado  
Italy  
October 1998

New in Csound version 3.49

# while

while — A syntactic looping construction.

## Description

A syntactic looping construction.

## Syntax

```
while condition do
... od
```

## Performance

The statements between the *do* and *od* form the body of a loop which is obeyed while the conditional remains true.

## Examples

Here is an example of the while construction. It uses the file *while.csd* [examples/while.csd].

### Example 1128. Example of the while opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o ifthen.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 1
lab99:
if p4<0 goto lab100
  p4 = p4-1
  print p4
  goto lab99
lab100:
endin

instr 2
  while p4>=0 do
    p4 = p4-1
    print p4
  od
```

```
endin
</CsInstruments>
<CsScore>
i 1 1 1 4
i 2 2 1 4
e

</CsScore>
</CsoundSynthesizer>
```

Its output should include lines like this:

```
B 0.000 .. 1.000 T 1.000 TT 1.000 M: 0.0
new alloc for instr 1:
instr 1: p4 = 3.000
instr 1: p4 = 2.000
instr 1: p4 = 1.000
instr 1: p4 = 0.000
instr 1: p4 = -1.000
B 1.000 .. 2.000 T 2.000 TT 2.000 M: 0.0
new alloc for instr 2:
instr 2: p4 = 3.000
instr 2: p4 = 2.000
instr 2: p4 = 1.000
instr 2: p4 = 0.000
instr 2: p4 = -1.000
B 2.000 .. 3.000 T 3.000 TT 3.000 M: 0.0
```

## See Also

*loop\_ge*, *loop\_gt* and *loop\_le*. *loop\_lt*. *until*.

## Credits

John fitch.

New in Csound version 6.04

# wiiconnect

wiiconnect — Reads data from a number of external Nintendo Wiimote controllers.

## Description

Opens and at control-rate polls up to four external Nintendo Wiimote controllers.

## Syntax

```
ires wiiconnect [itimeout, imaxnum]
```

## Initialization

*itimeout* -- integer number of seconds the system should wait for all Wiimotes to be connected. If not given it defaults to 10 seconds.

*imaxnum* -- maximum number of Wiimotes to locate. If not given it defaults to 4.

Initially each Wiimote has its numeric allocation indicated by lighting one of the four LEDs.

*ires* -- return value is 1 if success or zero on failure.

## Performance



### Note

Please note that these opcodes are currently only supported on Linux.

Every control cycle each Wiimote is polled for its status and position. These values are read by the *wiidata* opcode. The result returned is 1 in most cases, but will be zero if a Wiimote disconnects,

## Example

Here is an example of the wii opcodes. It uses the file *wii.csd* [examples/wii.csd].

### Example 1129. Example of the wii opcodes.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
+rtaudio=alsa -o dac:hw:0
</CsOptions>
<CsInstruments>
nchnls = 2
ksmps = 400

#define WII_B          #3#
#define WII_A          #4#
#define WII_R_A       #304#
```

```

#define WII_PITCH      #20#
#define WII_ROLL      #21#
#define WII_BATTERY   #27#

#define WII_RUMBLE    #3#
#define WII_SET_LEDS  #4#

gkcnt init 1

instr 1
  i1 wiiconnect 3,1

      wiirange  $WII_PITCH., -20, 0
kb wiidata    $WII_BATTERY.
kt wiidata    $WII_B.
ka wiidata    $WII_A.
kra wiidata  $WII_R.A.
gka wiidata  $WII_PITCH.
gkp wiidata  $WII_ROLL.
; If the B (trigger) button is pressed then activate a note
if (kt==0) goto ee
event "i", 2, 0, 5
gkcnt = gkcnt + 1
wiisend      $WII_SET_LEDS., gkcnt
ee:
  if (ka==0) goto ff
wiisend      $WII_RUMBLE., 1
ff:
  if (kra==0) goto gg
wiisend      $WII_RUMBLE., 0
gg:
  printk2 kb
endin

instr 2
  a1 oscil ampdbfs(gka), 440+gkp, 1
  outs a1, a1
endin

</CsInstruments>

<CsScore>
f1 0 4096 10 1
i1 0 300

</CsScore>

</CsoundSynthesizer>

```

## See Also

*wiidata, wiirange, wiisend*

## Credits

Author: John ffitc  
 Codemist Ltd  
 2009

New in version 5.11

# wiidata

wiidata — Reads data fields from a number of external Nintendo Wiimote controllers.

## Description

Reads data fields from upto four external Nintendo Wiimote controllers.

## Syntax

```
kres wiidata kcontrol[, knum]
```

## Initialization

This opcode must be used in conjunction with a running *wiiconnect* opcode.

## Performance



### Note

Please note that these opcodes are currently only supported on Linux.

*kcontrol* -- the code for which control to read

*knnum* -- the number of the which Wiimote to access, which defaults to the first one.

On each access a particular data item of the Wiimote is read. The currently implemented controls are given below, together with the macro name defined in the file *wii\_mac*:

0 (WII\_BUTTONS): returns a bit pattern for all buttons that were pressed.

1 (WII\_TWO): returns 1 if the button has just been pressed, or 0 otherwise.

2 (WII\_ONE): as above.

3 (WII\_B): as above.

4 (WII\_A): as above.

5 (WII\_MINUS): as above.

8 (WII\_HOME): as above.

9 (WII\_LEFT): as above.

10 (WII\_RIGHT): as above.

11 (WII\_DOWN): as above.

12 (WII\_UP): as above.

13 (WII\_PLUS): as above.

If the control number is 100 more than one of these button codes then the current state of the button is returned. Macros with names like `WII_S_TWO` etc are defined for this.

If the control number is 200 more than one of these button codes then the return value is 1 if the button is held and 0 otherwise. Macros with names like `WII_H_TWO` etc are defined for this.

If the control number is 300 more than one of these button codes then the value is 1 if the button has just been released, and 0 otherwise. Macros with names like `WII_R_TWO` etc are defined for this.

20 (`WII_PITCH`): The pitch of the Wiimote. The value is in degrees between -90 and +90, unless modified by a *wiirange* call.

21 (`WII_ROLL`): The roll of the Wiimote. The value is in degrees between -90 and +90, unless modified by a *wiirange* call.

23 (`WII_FORCE_X`): The force applied to the Wiimote in the three axes.

24 (`WII_FORCE_Y`):

25 (`WII_FORCE_Z`):

26 (`WII_FORCE_TOTAL`): The total magnitude of the force applied to the Wiimote.

27 (`WII_BATTERY`): The percent of the battery that remains.

28 (`WII_NUNCHUK_ANG`): The angle of the nunchuk joystick in degrees.

29 (`WII_NUNCHUK_MAG`): The magnitude of the nunchuk joystick from neutral.

30 (`WII_NUNCHUK_PITCH`): The pitch of the nunchuk in degrees, in range -90 to +90 unless modified by a *wiirange* call.

31 (`WII_NUNCHUK_ROLL`): The roll of the nunchuk in degrees, in range -90 to +90 unless modified by a *wiirange* call.

33 (`WII_NUNCHUK_Z`): The state of the nunchuk Z button.

34 (`WII_NUNCHUK_C`): The state of the nunchuk C button.

35 (`WII_IR1_X`): The infrared pointing of the Wiimote.

36 (`WII_IR1_Y`):

37 (`WII_IR1_Z`):

## Examples

See the example for *wiiconnect*.

## See Also

*wiiconnect*, *wiirange*, *wiisend*,

## Credits

Author: John ffitich

Codemist Ltd  
2009

New in version 5.11

# wiirange

wiirange — Sets scaling and range limits for certain Wiimote fields.

## Description

Sets scaling and range limits for certain Wiimote fields.

## Syntax

```
wiirange icontrol, iminimum, imaximum[, inum]
```

## Initialization

This opcode must be used in conjunction with a running *wiiconnect* opcode.

*icontrol* -- which control is to be scaled. This must be one of 20 (WII\_PITCH), 21 (WII\_ROLL), 30 (WII\_NUNCHUK\_PITCH), 31 (WII\_NUNCHUK\_ROLL).

*iminimum* -- minimum value for control.

*imaximum* -- maximum value for control.



### Note

Please note that these opcodes are currently only supported on Linux.

## Examples

See the example for *wiiconnect*.

## See Also

*wiiconnect*, *wiidata*, *wiisend*,

## Credits

Author: John ffitc  
Codemist Ltd  
2009

New in version 5.11

# wiisend

wiisend — Sends data to one of a number of external Nintendo Wiimote controllers.

## Description

Sends data to one of a number of external Nintendo Wiimote controllers.

## Syntax

```
kres wiisend kcontrol, kvalue[, knum]
```

## Initialization

This opcode must be used in conjunction with a running *wiiconnect* opcode.

## Performance



### Note

Please note that these opcodes are currently only supported on Linux.

*kcontrol* -- the code for which control to write.

*kvalue* -- the value to write to the control.

*knum* -- the number of the which Wiimote to access, which defaults to the first one (zero).

On each access a particular data item of the Wiimote is written. The currently implemented controls are given below, together with the macro name defined in the file *wii\_mac*:

3 (WII\_RUMBLE): starts or stops the Wiimote rumbling, depending on the value of *kvalue* (0 to stop, 1 to start).

4 (WII\_SET\_LEDS): set the four LED lights on the Wiimote to the binary representation of *kvalue*.

## Examples

See the example for *wiiconnect*.

## See Also

*wiiconnect*, *wiidata*, *wiirange*,

## Credits

Author: John ffitc  
Codemist Ltd  
2009

New in version 5.11

# window

window — Applies a window to an array.

## Description

Applies a given window shape to a vector stored in an array. The output is an array with the windowed vector.

## Initialization

*itype* -- optional window type: 0 = Hamming, 1 = Hanning (von Hann) (defaults to 1).

## Syntax

```
u
kout[] window kin[][, koff, itype]
```

## Performance

*kout[]* -- output array containing the windowed output. It will be created if it does not exist.

*kin[]* -- input array containing the input vector.

*koff* -- offset to make window start at position koff (non-negative only, defaults to 0).

## Examples

Here is an example of the window opcode. It uses the file *window.csd* [examples/window.csd].

### Example 1130. Example of the window opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>

<CsOptions>
-d -o dac
</CsOptions>

<CsInstruments>
;ksmps needs to be an integer div of hopsize
ksmps = 64

instr 1

  ihopsize = 256 ; hopsize
  ifftsize = 1024 ; FFT size
  iolaps = ifftsize/ihopsize ; overlaps
  ibw = sr/ifftsize ; bin bandwidth
  kcnt init 0 ; counting vars
  krow init 0

  kOla[] init ifftsize ; overlap-add buffer
  kIn[] init ifftsize ; input buffer
```

```

kOut[][] init iolaps, ifftsize ; output buffers

a1 diskin2 "fox.wav",1,0,1 ; audio input

/* every hopsize samples */
if kcnt == ihopsize then
  /* window and take FFT */
  kWin[] window kIn,krow*ihopsize
  kSpec[] rfft kWin

  /* filter between high and low freqs */
  ilow = 0
  ihigh = 1000
  ki = int(ilow/ibw)
  until ki == int(ihigh/ibw) do
    kSpec[ki] = 0
    ki += 1
  od

  /* IFFT + window */
  kRow[] rifft kSpec
  kWin window kRow, krow*ihopsize
  /* place it on out buffer */
  kOut setrow kWin, krow

  /* zero the ola buffer */
  kOla = 0
  /* overlap-add */
  ki = 0
  until ki == iolaps do
    kRow getrow kOut, ki
    kOla = kOla + kRow
    ki += 1
  od

  /* update counters */
  krow = (krow+1)%iolaps
  kcnt = 0
endif

/* shift audio in/out of buffers */
kIn shiftin a1
a2 shiftout kOla
out a2/iolaps

/* increment counter */
kcnt += ksmps

endin

</CsInstruments>

<CsScore>
i1 0 10
</CsScore>

</CsoundSynthesizer>

```

## See Also

Vectorial opcodes, array opcodes

## Credits

Author: Victor Lazzarini

NUI Maynooth  
2014

New in version 6.04

# wrap

wrap — Wraps-around the signal that exceeds the low and high thresholds.

## Description

Wraps-around the signal that exceeds the low and high thresholds.

## Syntax

```
ares wrap asig, klow, khigh
ires wrap isig, ilow, ihigh
kres wrap ksig, klow, khigh
```

## Initialization

```
isig -- input signal
ilow -- low threshold
ihigh -- high threshold
```

## Performance

```
xsig -- input signal
klow -- low threshold
khigh -- high threshold
```

*wrap* wraps-around the signal that exceeds the low and high thresholds.

This opcode is useful in several situations, such as table indexing or for clipping and modeling a-rate, i-rate or k-rate signals. *wrap* is also useful for wrap-around of table data when the maximum index is not a power of two (see *table* and *tablei*). Another use of *wrap* is in cyclical event repeating, with arbitrary cycle length.

## Examples

Here is an example of the wrap opcode. It uses the file *wrap.csd* [examples/wrap.csd].

### Example 1131. Example of the wrap opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
```

```

; -o wrap.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
Odbfs = 1
nchnls = 2

instr 1 ; Limit / Mirror / Wrap

igain = p4 ;gain
ilevl1 = p5 ; + level
ilevl2 = p6 ; - level
imode = p7 ;1 = limit, 2 = mirror, 3 = wrap

ain soundin "fox.wav"
ain = ain*igain

if imode = 1 goto limit
if imode = 2 goto mirror

asig wrap ain, ilevl2, ilevl1
goto outsignal

limit:
asig limit ain, ilevl2, ilevl1
goto outsignal

mirror:
asig mirror ain, ilevl2, ilevl1
outsignal:

outs asig*.5, asig*.5 ;mind your speakers

endin

</CsInstruments>
<CsScore>

;          Gain +Levl -Levl Mode
i1 0 3 4.00 .25 -1.00 1 ;limit
i1 4 3 4.00 .25 -1.00 2 ;mirror
i1 8 3 4.00 .25 -1.00 3 ;wrap
e
</CsScore>
</CsoundSynthesizer>

```

## See Also

*limit, mirror*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.49

# writescratch

writescratch — writes a value into the scratchpad of the instance of an instrument.

## Description

The *writescratch* opcode writes one of four scalar values to be stored in the instance of an instrument.

## Syntax

```
writescratchival[, index]
```

## Initialisation

*ival* -- variable to write.

*index* -- which value to write, defaulting to zero.

## Examples

Here is an example of the writescratch opcode. It uses the file *readscratch.csd* [examples/readscratch.csd].

### Example 1132. Example of the writescratch opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ilast readscratch
writescratch p2
ilen readscratch 1
writescratch p3, 1
printf_i "last run at %f for %f\n", ilen, ilast, ilen
endin

</CsInstruments>
<CsScore>
i 1 0 1
i 1 2 3
i 1 6 10
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*readscratch,*

## Credits

Author: John ffitch  
March 2013

New in Csound version 6.00

# wterrain

wterrain — A simple wave-terrain synthesis opcode.

## Description

A simple wave-terrain synthesis opcode.

## Syntax

```
aout wterrain kamp, kpch, k_xcenter, k_ycenter, k_xradius, k_yradius, \  
          itabx, itaby
```

## Initialization

*itabx*, *itaby* -- The two tables that define the terrain.

## Performance

The output is the result of drawing an ellipse with axes *k\_xradius* and *k\_yradius* centered at (*k\_xcenter*, *k\_ycenter*), and traversing it at frequency *kpch*.

## Examples

Here is an example of the wterrain opcode. It uses the file *wterrain.csd* [examples/wterrain.csd].

### Example 1133. Example of the wterrain opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
; Audio out  Audio in  No messages  
-odac          -iadc     -d      ;;RT audio I/O  
; For Non-realtime ouput leave only the line below:  
; -o wterrain.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>  
  
; Initialize the global variables.  
sr = 44100  
kr = 4410  
ksmps = 10  
nchnls = 1  
  
instr 1  
kdc1k linseg 0, 0.01, 1, p3-0.02, 1, 0.01, 0  
kcx line 0.1, p3, 1.9  
krx linseg 0.1, p3/2, 0.5, p3/2, 0.1  
kpch line cpspch(p4), p3, p5 * cpspch(p4)  
a1 wterrain 10000, kpch, kcx, kcx, -krx, krx, p6, p7  
a1 dcblock a1
```

```
      out      a1*kdclk
endin

</CsInstruments>
<CsScore>

f1      0      8192    10      1 0 0.33 0 0.2 0 0.14 0 0.11
f2      0      4096    10      1

i1      0      4       7.00 1 1 1
i1      4      4       6.07 1 1 2
i1      8      8       6.00 1 2 2
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Matthew Gillard  
New in version 4.19

## xadsr

`xadsr` — Calculates the classical ADSR envelope.

## Description

Calculates the classical ADSR envelope

## Syntax

```
ares xadsr iatt, idec, islev, irel [, idel]
```

```
kres xadsr iatt, idec, islev, irel [, idel]
```

## Initialization

*iatt* -- duration of attack phase

*idec* -- duration of decay

*islev* -- level for sustain phase

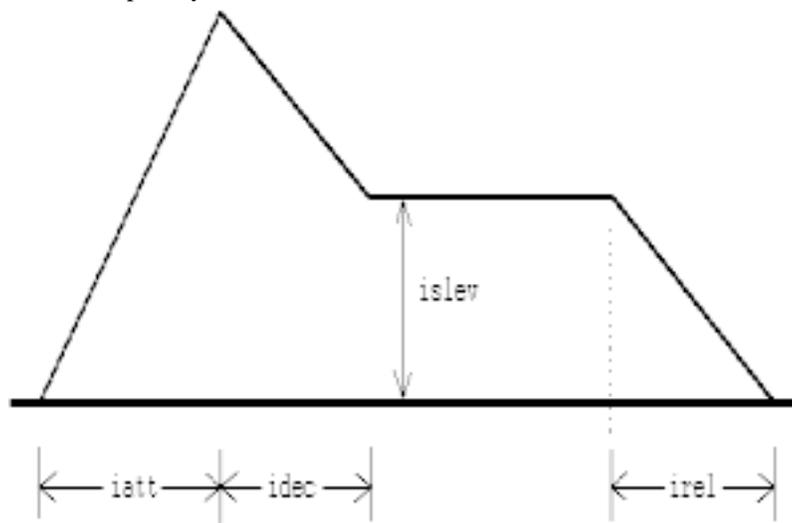
*irel* -- duration of release phase

*idel* -- period of zero before the envelope starts

## Performance

The envelope generated is the range 0 to 1 and may need to be scaled further, depending on the amplitude required. If using *Odbfs* = 1, scaling down will probably be required since playing more than one note might result in clipping. If not using *Odbfs*, scaling to a large amplitude (e.g. 32000) might be required.

The envelope may be described as:



Picture of an ADSR envelope.

The length of the sustain is calculated from the length of the note. This means *xadsr* is not suitable for use with MIDI events, use *mxadsr* instead. The opcode *xadsr* is identical to *adsr* except it uses exponential, rather than linear, line segments.

## Examples

Here is an example of the *xadsr* opcode. It uses the file *xadsr.csd* [examples/xadsr.csd].

### Example 1134. Example of the *xadsr* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o xadsr.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

iatt = p5
idec = p6
islev = p7
irel = p8

kenv xadsr iatt, idec, islev, irel
kcps = cpspch(p4) ;frequency

asig vco2 kenv * 0.8, kcps
outs asig, asig

endin

</CsInstruments>
<CsScore>

i 1 0 1 7.00 .0001 1 .01 .001 ; short attack
i 1 2 1 7.02 1 .5 .01 .001 ; long attack
i 1 4 2 6.09 .0001 1 .1 .7 ; long release

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

*adsr*, *madsr*, *mxadsr*

## Credits

Author: John ffitch *xadsr* is new in Csound version 3.51.

# xin

`xin` — Passes variables to a user-defined opcode block,

## Description

The `xin` and `xout` opcodes copy variables to and from the opcode definition, allowing communication with the calling instrument.

The types of input and output variables are defined by the parameters *intypes* and *outtypes*.



### Notes

- `xin` and `xout` should be called only once, and `xin` should precede `xout`, otherwise an init error and deactivation of the current instrument may occur.
- These opcodes actually run only at i-time. Performance time copying is done by the user opcode call. This means that skipping `xin` or `xout` with `kgoto` has no effect, while skipping with `igoto` affects both init and performance time operation.

## Syntax

```
xinarg1 [, xinarg2] ... [xinargN] xin
```

## Performance

`xinarg1`, `xinarg2`, ... - input arguments. The number and type of variables must agree with the user-defined opcode's *intypes* declaration. However, `xin` does not check for incorrect use of init-time and control-rate variables.

The syntax of a user-defined opcode block is as follows:

```
opcode name, outtypes, intypes
xinarg1 [, xinarg2] [, xinarg3] ... [xinargN] xin
[setksmps iksmps]
... the rest of the instrument's code.
xout xoutarg1 [, xoutarg2] [, xoutarg3] ... [xoutargN]
endop
```

The new opcode can then be used with the usual syntax:

```
[xinarg1] [, xinarg2] ... [xinargN] name [xoutarg1] [, xoutarg2] ... [xoutargN] [, iksmps]
```

## Examples

See the example for the `opcode` opcode.

## See Also

`endop`, `opcode`, `setksmps`, `xout`

## Credits

Author: Istvan Varga, 2002; based on code by Matt J. Ingalls

New in version 4.22

## xout

`xout` — Retrieves variables from a user-defined opcode block,

## Description

The `xin` and `xout` opcodes copy variables to and from the opcode definition, allowing communication with the calling instrument.

The types of input and output variables are defined by the parameters *intypes* and *outtypes*.



### Notes

- `xin` and `xout` should be called only once, and `xin` should precede `xout`, otherwise an init error and deactivation of the current instrument may occur.
- These opcodes actually run only at i-time. Performance time copying is done by the user opcode call. This means that skipping `xin` or `xout` with `kgoto` has no effect, while skipping with `igoto` affects both init and performance time operation.

## Syntax

```
xout xoutarg1 [, xoutarg2] ... [, xoutargN]
```

## Performance

`xoutarg1`, `xoutarg2`, ... - output arguments. The number and type of variables must agree with the user-defined opcode's *outtypes* declaration. However, `xout` does not check for incorrect use of init-time and control-rate variables.

The syntax of a user-defined opcode block is as follows:

```
opcode name, outtypes, intypes
xinarg1 [, xinarg2] [, xinarg3] ... [xinargN] xin
[setksmps iksmps]
... the rest of the instrument's code.
xout xoutarg1 [, xoutarg2] [, xoutarg3] ... [xoutargN]
endop
```

The new opcode can then be used with the usual syntax:

```
[xinarg1] [, xinarg2] ... [xinargN] name [xoutarg1] [, xoutarg2] ... [xoutargN] [, iksmps]
```

## Examples

See the example for the `opcode` opcode.

## See Also

`endop`, `opcode`, `setksmps`, `xin`

## Credits

Author: Istvan Varga, 2002; based on code by Matt J. Ingalls

New in version 4.22

# xscanmap

`xscanmap` — Allows the position and velocity of a node in a scanned process to be read.

## Description

Allows the position and velocity of a node in a scanned process to be read.

## Syntax

```
kpos, kvel xscanmap iscan, kamp, kvamp [, iwhich]
```

## Initialization

`iscan` -- which scan process to read

`iwhich` (optional) -- which node to sense. The default is 0.

## Performance

`kamp` -- amount to amplify the `kpos` value.

`kvamp` -- amount to amplify the `kvel` value.

The internal state of a node is read. This includes its position and velocity. They are amplified by the `kamp` and `kvamp` values.

## Examples

Here is an example of the `xscanmap` opcode. It uses the file `xscanmap.csd` [examples/xscanmap.csd].

### Example 1135. Example of the `xscanmap` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o xscanmap.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
;the matrices can be found in /manual/examples
```

```

instr 1 ; Plain scanned syntnesis
; note - scanu display is turned off
a0 = 0
    xscanu 1, .01, 6, 2, "128-stringcircularX", 4, 5, 2, .1, .1, -.01, .1, .5, 0, 0, a0, 0, 0
a1 xscans p4, cpspch(p5), 7, 0, 3
k1,k2 xscanmap 0, 1000, 1000, 64
    display k1, .25 ; note - display is updated every second
    outs a1, a1
endin

instr 2 ; Scan synthesis with audio injection and dual scan paths
; note - scanu display is turned off
ain diskin2 "fox.wav",1,0,1
ain in
a0 = ain/10000
    xscanu 1, .01, 6, 2, "128,8-gridX", 14, 5, 2, .01, .05, -.05, .1, .5, 0, 0, a0, 0, 0
a1 xscans p4, cpspch(p5), 7, 0, 2
a2 xscans p4, cpspch(p6), 77, 0, 3
k1,k2 xscanmap 0, 1000, 1000, 127
    display k2, .5 ; note - display is updated ever 500ms
    outs a1,a2
endin

</CsInstruments>
<CsScore>
; Initial condition
;f1 0 16 7 0 8 1 8 0
f1 0 128 7 0 64 1 64 0

; Masses
f2 0 128 -7 1 128 1

; Centering force
f4 0 128 -7 0 128 2
f14 0 128 -7 2 64 0 64 2

; Damping
f5 0 128 -7 1 128 1

; Initial velocity
f6 0 128 -7 -.0 128 .0

; Trajectories
f7 0 128 -5 .001 128 128
f77 0 128 -23 "128-spiral-8,16,128,2,lover2"

; Sine
f9 0 1024 10 1
;-----
; Note list
i1 0 10 .9 7.00
s
i2 0 10 1 8.00 6.00
i2 0 10 1 7.00 8.05
e
</CsScore>
</CsSoundSynthesizer>

```

## See Also

More information on Scanned Synthesis (as well as several other matrices) is available on the *Scanned Synthesis page* [<http://www.csounds.com/scanned/>] at cSounds.com.

Also an article on these opcodes: [http://www.csounds.com/stevenyi/scanned/yi\\_scannedSynthesis.html](http://www.csounds.com/stevenyi/scanned/yi_scannedSynthesis.html) , written by Steven Yi

## Credits

Author: John ffitch

New in version 4.20

## xscansmap

`xscansmap` — Allows the position and velocity of a node in a scanned process to be read.

### Description

Allows the position and velocity of a node in a scanned process to be read.

### Syntax

```
xscansmap kpos, kvel, iscan, kamp, kvamp [, iwhich]
```

### Initialization

`iscan` -- which scan process to read

`iwhich` (optional) -- which node to sense. The default is 0.

### Performance

`kpos` -- the node's position.

`kvel` -- the node's velocity.

`kamp` -- amount to amplify the `kpos` value.

`kvamp` -- amount to amplify the `kvel` value.

The internal state of a node is read. This includes its position and velocity. They are amplified by the `kamp` and `kvamp` values.

### See Also

More information on Scanned Synthesis (as well as several other matrices) is available on the *Scanned Synthesis page* [<http://www.csounds.com/scanned/>] at cSounds.com.

Also an article on these opcodes: [http://www.csounds.com/stevenyi/scanned/yi\\_scannedSynthesis.html](http://www.csounds.com/stevenyi/scanned/yi_scannedSynthesis.html) , written by Steven Yi

### Credits

New in version 4.21

November 2002. Thanks to Rasmus Ekman for pointing this opcode out.

## xscans

xscans — Fast scanned synthesis waveform and the wavetable generator.

### Description

Experimental version of *scans*. Allows much larger matrices and is faster and smaller but removes some (unused?) flexibility. If liked, it will replace the older opcode as it is syntax compatible but extended.

### Syntax

```
ares xscans kamp, kfreq, ifntraj, id [, iorder]
```

### Initialization

*ifntraj* -- table containing the scanning trajectory. This is a series of numbers that contains addresses of masses. The order of these addresses is used as the scan path. It should not contain values greater than the number of masses, or negative numbers. See the *introduction to the scanned synthesis section*.

*id* -- If positive, the ID of the opcode. This will be used to point the scanning opcode to the proper waveform maker. If this value is negative, the absolute of this value is the wavetable on which to write the waveshape. That wavetable can be used later from an other opcode to generate sound. The initial contents of this table will be destroyed.

*iorder* (optional, default=0) -- order of interpolation used internally. It can take any value in the range 1 to 4, and defaults to 4, which is quartic interpolation. The setting of 2 is quadratic and 1 is linear. The higher numbers are slower, but not necessarily better.

### Performance

*kamp* -- output amplitude. Note that the resulting amplitude is also dependent on instantaneous value in the wavetable. This number is effectively the scaling factor of the wavetable.

*kfreq* -- frequency of the scan rate

### Matrix Format

The new matrix format is a list of connections, one per line linking point x to point y. There is no weight given to the link; it is assumed to be unity. The list is preceded by the line <MATRIX> and ends with a </MATRIX> line

For example, a circular string of 8 would be coded as

```
<MATRIX>
0 1
1 0
1 2
2 1
2 3
3 2
3 4
4 3
4 5
5 4
5 6
```

```

6 5
6 7
7 6
0 7
</MATRIX>

```

## Examples

Here is an example of the `xscans` opcode. It uses the file `xscans.csd` [examples/xscans.csd].

### Example 1136. Example of the `xscans` opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o xscans.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
Odbfs = 1
;the matrices can be found in /manual/examples

instr 1 ; Plain scanned syntnesis

a0      =      0
xscanu  1, .01, 6, 2, "128,8-cylinderX", 4, 5, 2, .1, .1, -.01, .1, .5, 0, 0, a0, 0, 0
a1      xscans .7, cpspch(p4), 7, 0, 1
outs    a1, a1

endin

instr 2 ; Scan synthesis with audio injection and dual scan paths
; tap the mic or sing to inject audio into the resonators

a0,aa   ins
a0      =      a0/.8
xscanu  1, .01, 6, 2, "128,8-gridX", 14, 5, 2, .01, .05, -.05, .1, .5, 0, 0, a0, 0, 0
a1      xscans .5, cpspch(7.00), 7, 0, 1
a2      xscans .5, cpspch(7.001), 77, 0, 1
outs    a1+a2,a1+a2

endin

instr 3 ; Vibrating structure with audio injection
; Tap the MIC - to inject audio into the resonators

a0,aa   ins
a0      =      a0/.8
xscanu  1, .01, 6, 2, "128-stringcircularX", 14, 5, 2, .01, .05, -.05, .25, .75, 0, 0, a0, 0, 0

endin

instr 4 ; Modulated scanners

```

```

i1      bexprnd  5
i2      bexprnd  1
ko      oscil   i1, i2, 9
ka1     oscili  .5, .15*8, p7
ka2     oscili  .5, .15*8, p8
kf      oscili  1, .15, p4
kf      =       2^(kf/12)*p6*440+ko
a1      xscans  p9*ka1, kf+i1, 777, 1, 1
a2      xscans  p9*ka2, (kf+i1)*2.1, 77, 1, 1
        outs    a1+a2, a1+a2

endin

</CsInstruments>
<CsScore>
; Initial condition
f1 0 128 7 0 64 1 64 0
; Masses
f2 0 128 -7 1 128 1
; Centering force
f4 0 128 -7 0 128 2
f14 0 128 -7 2 64 0 64 2
; Damping
f5 0 128 -7 1 128 1
; Initial velocity
f6 0 128 -7 -.0 128 .0
; Trajectories
f7 0 128 -5 .001 128 128
f777 0 128 -23 "128-stringcircular"
f77 0 128 -23 "128-spiral-8,16,128,2,lover2"
; Sine
f9 0 16384 10 1

; Pitch tables
f100 0 1024 -7 +3 128 +3 128 -2 128 -2 128 +0 128 +0 128 -4 128 -4 128 +3
f101 0 1024 -7 -2 128 -2 128 -2 128 -2 128 -5 128 -5 128 -4 128 -4 128 -2
f102 0 1024 -7 +3 128 +3 128 +2 128 +2 128 +0 128 +0 128 +0 128 +0 128 +3
f103 0 1024 -7 +7 128 +7 128 +5 128 +5 128 +3 128 +3 128 +3 128 +3 128 +7

; Amplitude tables
f200 0 1024 7 1 128 0 128 0 127 0 1 1 128 0 128 0 127 0 1 1 128 0 127 0 1 1
f201 0 1024 7 0 127 0 1 1 127 0 1 1 128 0 127 0 1 1 127 0 1 1 128 0 127 0 1 1
f202 0 1024 7 1 127 0 1 1 127 0 1 1 127 0 1 1 127 0 1 1 127 0 1 1 127 0 1 1
f203 0 1024 7 1 1024 0

;-----

; Note list
i1 0 10 6.00
s
i2 1 10
s
i3 1 23
i4 1 23 101 1 .5 200 202 1.5
i4 . . 102 0 .5 200 201 1
i4 . . 103 0 .5 200 201 1
i4 . . 100 0 .25 200 200 2
e
</CsScore>
</CsSoundSynthesizer>

```

For similar examples, see the documentation on *scans*.

## See Also

More information on Scanned Synthesis (as well as several other matrices) is available on the *Scanned Synthesis page* [<http://www.csounds.com/scanned/>] at cSounds.com.

Also an article on these opcodes: [http://www.csounds.com/stevenyi/scanned/yi\\_scannedSynthesis.html](http://www.csounds.com/stevenyi/scanned/yi_scannedSynthesis.html) ,  
written by Steven Yi

*scans, xscanu*

## Credits

Written by John ffitich.

New in version 4.20

## xscanu

xscanu — Compute the waveform and the wavetable for use in scanned synthesis.

### Description

Experimental version of *scanu*. Allows much larger matrices and is faster and smaller but removes some (unused?) flexibility. If liked, it will replace the older opcode as it is syntax compatible but extended.

### Syntax

```
xscanu init, irate, ifnvel, ifnmass, ifnstif, ifncentr, ifndamp, kmass, \  
kstif, kcentr, kdamp, ileft, iright, kpos, kstrngth, ain, idisp, id
```

### Initialization

*init* -- the initial position of the masses. If this is a negative number, then the absolute of *init* signifies the table to use as a hammer shape. If *init* > 0, the length of it should be the same as the intended mass number, otherwise it can be anything.

*irate* -- update rate.

*ifnvel* -- the ftable that contains the initial velocity for each mass. It should have the same size as the intended mass number.

*ifnmass* -- ftable that contains the mass of each mass. It should have the same size as the intended mass number.

*ifnstif* --

- *either* an ftable that contains the spring stiffness of each connection. It should have the same size as the square of the intended mass number. The data ordering is a row after row dump of the connection matrix of the system.
- *or* a string giving the name of a file in the MATRIX format

*ifncentr* -- ftable that contains the centering force of each mass. It should have the same size as the intended mass number.

*ifndamp* -- the ftable that contains the damping factor of each mass. It should have the same size as the intended mass number.

*ileft* -- If *init* < 0, the position of the left hammer (*ileft* = 0 is hit at leftmost, *ileft* = 1 is hit at rightmost).

*iright* -- If *init* < 0, the position of the right hammer (*iright* = 0 is hit at leftmost, *iright* = 1 is hit at rightmost).

*idisp* -- If 0, no display of the masses is provided.

*id* -- If positive, the ID of the opcode. This will be used to point the scanning opcode to the proper waveform maker. If this value is negative, the absolute of this value is the wavetable on which to write the waveshape.

That wavetable can be used later from an other opcode to generate sound. The initial contents of this table will be destroyed.

## Performance

*kmass* -- scales the masses

*kstif* -- scales the spring stiffness

*kcentr* -- scales the centering force

*kdamp* -- scales the damping

*kpos* -- position of an active hammer along the string (*kpos* = 0 is leftmost, *kpos* = 1 is rightmost). The shape of the hammer is determined by *init* and the power it pushes with is *kstrngth*.

*kstrngth* -- power that the active hammer uses

*ain* -- audio input that adds to the velocity of the masses. Amplitude should not be too great.

## Matrix Format

The new matrix format is a list of connections, one per line linking point *x* to point *y*. There is no weight given to the link; it is assumed to be unity. The list is preceded by the line `<MATRIX>` and ends with a `</MATRIX>` line

For example, a circular string of 8 would be coded as

```
<MATRIX>
0 1
1 0
1 2
2 1
2 3
3 2
3 4
4 3
4 5
5 4
5 6
6 5
6 7
7 6
0 7
</MATRIX>
```

## Examples

Here is an example of the `xscanu` opcode. It uses the file `xscanu.csd` [examples/xscanu.csd].

### **Example 1137. Example of the `xscanu` opcode.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o xscanu.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
Odbfs = 1
;the matrices can be found in /manual/examples

instr 1 ; Plain scanned syntnesis
; Note Also that I am using quadratic interpolation on these.
a0 = 0
xscanu 1, .01, 6, 2, "128,8-gridX", 4, 5, 2, .1, .1, -.01, .1, .5, 0, 0, a0, 0, 0
a1 xscans .5, cpspch(p4), 333, 0, p6 ; NOTE LEFT RIGHT TRAJECTORY (f333) IS CLEAN!
a1 dcblock a1
outs a1, a1
endin

instr 2 ; Scan synthesis with audio injection and dual scan paths

a0 diskin2 "fox.wav",1,0,1
; a0,aa ins
a0 = a0/.8
xscanu 1, .01, 6, 2, "128,8-torusX", 14, 5, 2, .01, .05, -.05, .1, .5, 0, 0, a0, 0, 0
a1 xscans .3, cpspch(7.00), 333, 0, 2 ; NOTE LEFT RIGHT TRAJECTORY (f333) IS CLEAN!
a2 xscans .3, cpspch(6.00), 77, 0, 2
a1 dcblock a1
a2 dcblock a2
outs a1*.5,a2*.1
endin

</CsInstruments>
<CsScore>

; Initial condition
;f1 0 16 7 0 8 1 8 0
f1 0 128 7 0 64 1 64 0

; Masses
f2 0 128 -7 1 128 1

; Centering force
f4 0 128 -7 0 128 2
f14 0 128 -7 2 64 0 64 2

; Damping
f5 0 128 -7 1 128 1

; Initial velocity
f6 0 128 -7 -.0 128 .0

; Trajectories
f7 0 128 -5 .001 128 128
f77 0 128 -23 "128-spiral-8,16,128,2,lover2"
f777 0 128 -23 "128,8-torusX"

; Spring matrices

```

```
f3 0 128 -23 "128-stringX"
f33 0 128 -23 "128-stringcircularX"
f333 0 128 -23 "128-left_rightX"
f3333 0 128 -23 "128,8-torusX"
f33333 0 128 -23 "128,8-cylinderX"
f333333 0 128 -23 "128,8-gridX"

; Sine
f9 0 1024 10 1

; Pitch tables
f100 0 1024 -7 +3 128 +3 128 -2 128 -2 128 +0 128 +0 128 -4 128 -4 128 +3
f101 0 1024 -7 -2 128 -2 128 -2 128 -2 128 -5 128 -5 128 -4 128 -4 128 -2
f102 0 1024 -7 +3 128 +3 128 +2 128 +2 128 +0 128 +0 128 +0 128 +3
f103 0 1024 -7 +7 128 +7 128 +5 128 +5 128 +3 128 +3 128 +3 128 +3 128 +7

; Amplitude tables
f200 0 1024 7 1 128 0 128 0 127 0 1 1 128 0 128 0 127 0 1 1 128 0 127 0 1 1
f201 0 1024 7 0 127 0 1 1 127 0 1 1 128 0 127 0 1 1 127 0 1 1 128 0 127 0 1 1 127 0 1 1
f202 0 1024 7 1 127 0 1 1 127 0 1 1 127 0 1 1 127 0 1 1 127 0 1 1 127 0 1 1 127 0 1 1 127 0 1
f203 0 1024 7 1 1024 0
;f204 0 1024 7 1 512 0 511 0 1 1

;-----

; Note list
i1 0 10 6.00 1 2
s
i2 0 15
e
</CsScore>
</CsoundSynthesizer>
```

For similar examples, see the documentation on *scans*.

## See Also

More information on Scanned Synthesis (as well as several other matrices) is available on the *Scanned Synthesis page* [<http://www.csounds.com/scanned/>] at cSounds.com.

Also an article on these opcodes: [http://www.csounds.com/stevenyi/scanned/yi\\_scannedSynthesis.html](http://www.csounds.com/stevenyi/scanned/yi_scannedSynthesis.html) , written by Steven Yi

*scanu*, *xscans*

## Credits

Written by John ffitc.

New in version 4.20

# xtratim

xtratim — Extend the duration of real-time generated events.

## Description

Extend the duration of real-time generated events and handle their extra life (Usually for usage along with *release* instead of *linenr*, *linsegr*, etc).

## Syntax

```
xtratim iextradur
```

## Initialization

*iextradur* -- additional duration of current instrument instance

## Performance

*xtratim* extends current MIDI-activated note duration by *iextradur* seconds after the corresponding noteoff message has deactivated the current note itself. It is usually used in conjunction with *release*. This opcode has no output arguments.

This opcode is useful for implementing complex release-oriented envelopes, whose duration is not known when the envelope starts (e.g. for real-time MIDI generated events).

## Examples

Here is a simple example of the xtratim opcode. It uses the file *xtratim.csd* [examples/xtratim.csd].

### Example 1138. Example of the xtratim opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

This example shows how to generate a release segment for an ADSR envelope after a MIDI noteoff is received, extending the duration with *xtratim* and using *release* to check whether the note is on the release phase.

```
<CsoundSynthesizer>

<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  Silent MIDI in
-odac        -iadc     -d         -M0   ;;realtime I/O
</CsOptions>

<CsInstruments>
;Simple usage of the xtratim opcode
sr = 44100
ksmps = 10
nchnls = 2

; sine wave for oscillators
gisin      ftgen      1, 0, 4096, 10, 1
```

```

instr 1

  inum notnum
  icps cpsmidi
  iamp ampmidi 4000
  ;
  ;----- complex envelope block -----
  xtratim 1 ;extra-time, i.e. release dur
  krel init 0
  krel release ;outputs release-stage flag (0 or 1 values)
  if (krel == 1) kgoto rel ;if in release-stage goto release section
  ;
  ;***** attack and sustain section *****
  kmp1 linseg 0, .03, 1, .05, 1, .07, 0, .08, .5, 4, 1, 50, 1
  kmp = kmp1*iamp
  kgoto done
  ;
  ;----- release section -----
  rel:
  kmp2 linseg 1, .3, .2, .7, 0
  kmp = kmp1*kmp2*iamp
  done:
  ;-----
  a1 oscili kmp, icps, gisin
  outs a1, a1
  endin

</CsInstruments>

<CsScore>
f 0 3600 ;dummy table to wait for realtime MIDI events
e
</CsScore>

</CsoundSynthesizer>

```

Here is a more elaborate example of the *xtratim* opcode. It uses the file *xtratim-2.csd* [examples/xtratim-2.csd].

### Example 1139. More complex example of the *xtratim* opcode.

This example shows how to generate a release segment for an ADSR envelope after a MIDI noteoff is received, extending the duration with *xtratim* and using *release* to check whether the note is on the release phase. Two envelopes are generated simultaneously for the left and right channels.

```

<CsoundSynthesizer>

<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  Silent  MIDI in
-odac          -iadc    -d      -MO   ;;realtime I/O
</CsOptions>

<CsInstruments>
;xtratim example by Jonathan Murphy Dec. 2006
sr = 44100
ksmps = 10
nchnls = 2

  ; sine wave for oscillators
  gisin ftgen 1, 0, 4096, 10, 1
  ; set volume initially to midpoint
  ctrlinit 1, 7,64

  ;; simple two oscil, two envelope synth

```

```

instr 1

    ; frequency
kcps    cpsmidib
    ; initial velocity (noteon)
ivel    veloc

    ; master volume
kamp    ctrl17    1, 7, 0, 127
kamp    = kamp * ivel

    ; parameters for aenv1
iatt1   = 0.03
idec1   = 1
isus1   = 0.25
irel1   = 1
    ; parameters for aenv2
iatt2   = 0.06
idec2   = 2
isus2   = 0.5
irel2   = 2

    ; extra (release) time allocated
xtratim (irel1>irel2 ? irel1 : irel2)
    ; krel is used to trigger envelope release
krel    init    0
krel    release
    ; if noteoff received, krel == 1, otherwise krel == 0
if (krel == 1) kgoto rel

    ; attack, decay, sustain segments
atmp1   linseg    0, iatt1, 1, idec1, isus1 , 1, isus1
atmp2   linseg    0, iatt2, 1, idec2, isus2 , 1, isus2
aenv1   = atmp1
aenv2   = atmp2
kgoto   done

    ; release segment
rel:
atmp3   linseg    1, irel1, 0, 1, 0
atmp4   linseg    1, irel2, 0, 1, 0
aenv1   = atmp1 * atmp3 ;to go from the current value (in case
aenv2   = atmp2 * atmp4 ;the attack hasn't finished) to the release.

    ; control oscillator amplitude using envelopes
done:
aoscl   oscil    aenv1, kcps, gisin
aoscr   oscil    aenv2, kcps * 1.5, gisin
aoscl   = aoscl * kamp
aoscr   = aoscr * kamp

    ; send aoscl to left channel, aoscr to right,
    ; release times are noticeably different

outs    aoscl, aoscr

endin

</CsInstruments>

<CsScore>

f 0 3600 ;dummy table to wait for realtime MIDI events

</CsScore>

</CsoundSynthesizer>

```

## See Also

*linenr, release*

## Credits

Author: Gabriel Maldonado

Italy

Examples by Gabriel Maldonado and Jonathan Murphy

New in Csound version 3.47

# xyin

xyin — Sense the cursor position in an output window

## Description

Sense the cursor position in an output window. When *xyin* is called the position of the mouse within the output window is used to reply to the request. This simple mechanism does mean that only one *xyin* can be used accurately at once. The position of the mouse is reported in the output window.

## Syntax

```
kx, ky xyin iprd, ixmin, ixmax, ymin, ymax [, ixinit] [, iyinit]
```

## Initialization

*iprd* -- period of cursor sensing (in seconds). Typically .1 seconds.

*xmin*, *xmax*, *ymin*, *ymax* -- edge values for the x-y coordinates of a cursor in the input window.

*ixinit*, *iyinit* (optional) -- initial x-y coordinates reported; the default values are 0,0. If these values are not within the given min-max range, they will be coerced into that range.

## Performance

*xyin* samples the cursor x-y position in an input window every *iprd* seconds. Output values are repeated (not interpolated) at the k-rate, and remain fixed until a new change is registered in the window. There may be any number of input windows. This unit is useful for real-time control, but continuous motion should be avoided if *iprd* is unusually small.



### Note

Depending on your platform and distribution, you might need to enable displays using the *--displays* command line flag.

## Examples

Here is an example of the *xyin* opcode. It uses the file *xyin.csd* [examples/xyin.csd].

### Example 1140. Example of the *xyin* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      --displays ;;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o xyin.wav -W ;;; for file output any platform
```

```
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
ksmps = 10
nchnls = 2

; Instrument #1.
instr 1
; Print and capture values every 0.1 seconds.
iprd = 0.1
; The x values are from 1 to 30.
ixmin = 1
ixmax = 30
; The y values are from 1 to 30.
iymin = 1
iymax = 30
; The initial values for X and Y are both 15.
ixinit = 15
iyinit = 15

; Get the values kx and ky using the xyin opcode.
kx, ky xyin iprd, ixmin, ixmax, iymin, iymax, ixinit, iyinit

; Print out the values of kx and ky.
printks "kx=%f, ky=%f\n", iprd, kx, ky

; Play an oscillator, use the x values for amplitude and
; the y values for frequency.
kamp = kx * 1000
kcps = ky * 220
a1 poscil kamp, kcps, 1

outs a1, a1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for 30 seconds.
i 1 0 30
e

</CsScore>
</CsSoundSynthesizer>
```

As the values of kx and ky change, they will be printed out like this:

```
kx=8.612036, ky=22.677933
kx=10.765685, ky=15.644135
```

## Credits

Example written by Kevin Conder.

# xyscale

xyscale — 2D linear interpolation

## Description

2D linear interpolation between 4 points at (0,0), (1,0), (0,1), (1,1)

## Syntax

```
kout xyscale kx, ky, k00, k10, k01, k11
```

## Performance

*kx, ky* -- Coordinates to evaluate the interpolation. Value between 0-1, where:

*k00* -- Value of point at coord (x=0, y=0)

*k10* -- Value of point at coord (x=1, y=0)

*k01* -- Value of point at coord (x=0, y=1)

*k11* -- Value of point at coord (x=1, y=1)

```
(0,1)      (1,1)
(0,0)      (1,0)
```

Given 4 values placed at the corners of a square, find the interpolated value at point (x, y), where x and y are defined between 0-1.

## Examples

Here is an example of the xyscale opcode. It uses the file *xyscale.csd* [examples/xyscale.csd].

### Example 1141. Example of the xyscale opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc      -d          ;;RT audio I/O
</CsOptions>
<CsInstruments>

ksmps=128
nchnls=2
```

```
giwidth = 400
giheight = 300
FLpanel "FLmouse", giwidth, giheight, 10, 10
FLpanelEnd

FLrun

Odbfs = 1

instr 1
; We define four chords for bottom-left, bottom-right, top-left and top-right
; Use the mouse to interpolate between them
ibl[] array ntom:i("4C"), ntom:i("4Eb"), ntom:i("4G")
itl[] array ntom:i("4E"), ntom:i("4G#"), ntom:i("4B")
ibr[] array ntom:i("4G"), ntom:i("4A"), ntom:i("4B")
itr[] array ntom:i("4Eb"), ntom:i("4Eb+"), ntom:i("4F")

kmousex, kmousey, kb1, kb2, kb3    FLmouse 2
kx = limit(kmousex/giwidth, 0, 1)
ky = 1 - limit(kmousey/giheight, 0, 1)

printf "x: %f   y: %f \n", changed(kx, ky), kx, ky

iamp = 0.1
a0 oscili iamp, mtof(xyscale(kx, ky, ibl[0], itl[0], ibr[0], itr[0]))
a1 oscili iamp, mtof(xyscale(kx, ky, ibl[1], itl[1], ibr[1], itr[1]))
a2 oscili iamp, mtof(xyscale(kx, ky, ibl[2], itl[2], ibr[2], itr[2]))
aout = sum(a0, a1, a2)
outs aout, aout
endin

</CsInstruments>
<CsScore>
i 1 0 120

</CsScore>
</CsoundSynthesizer>
```

## See Also

*scale, ntrpol, linlin*

## Credits

By: Eduardo Moguillansky 2017

# zacl

zacl — Clears one or more variables in the za space.

## Description

Clears one or more variables in the za space.

## Syntax

```
zacl kfirst, klast
```

## Performance

*kfirst* -- first zk or za location in the range to clear.

*klast* -- last zk or za location in the range to clear.

*zacl* clears one or more variables in the za space. This is useful for those variables which are used as accumulators for mixing a-rate signals at each cycle, but which must be cleared before the next set of calculations.

## Examples

Here is an example of the *zacl* opcode. It uses the file *zacl.csd* [examples/zacl.csd].

### Example 1142. Example of the *zacl* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc    -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o zacl.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1

; Instrument #1 -- a simple waveform.
instr 1
; Generate a simple sine waveform.
asin oscil 20000, 440, 1
```

```
    ; Send the sine waveform to za variable #1.
    zaw asin, 1
  endin

  ; Instrument #2 -- generates audio output.
  instr 2
    ; Read za variable #1.
    a1 zar 1

    ; Generate the audio output.
    out a1

    ; Clear the za variables, get them ready for
    ; another pass.
    zacl 0, 1
  endin

</CsInstruments>
<CsScore>

  ; Table #1, a sine wave.
  f 1 0 16384 10 1

  ; Play Instrument #1 for one second.
  i 1 0 1
  ; Play Instrument #2 for one second.
  i 2 0 1
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*zamod, zar, zaw, zawm, ziw, ziwM*

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.45

Example written by Kevin Conder.

# zakinit

zakinit — Establishes zak space.

## Description

Establishes zak space. Must be called only once.

## Syntax

```
zakinit isizea, isizek
```

## Initialization

*isizea* -- the number of audio rate locations for a-rate patching. Each location is actually an array which is *ksmps* long.

*isizek* -- the number of locations to reserve for floats in the zk space. These can be written and read at i- and k-rates.

## Performance

At least one location each is always allocated for both za and zk spaces. There can be thousands or tens of thousands za and zk ranges, but most pieces probably only need a few dozen for patching signals. These patching locations are referred to by number in the other zak opcodes.

To run *zakinit* only once, put it outside any instrument definition, in the orchestra file header, after *sr*, *kr*, *ksmps*, and *nchnls*.



### Note

Zak channels count from 0, so if you define 1 channel, the only valid channel is channel 0.

## Examples

Here is an example of the *zakinit* opcode. It uses the file *zakinit.csd* [examples/zakinit.csd].

### Example 1143. Example of the *zakinit* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o zakinit.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 4410
nchnls = 1
```

```

; Initialize the ZAK space.
; Create 3 a-rate variables and 5 k-rate variables.
zakinit 2, 3

instr 1 ;a simple waveform.
; Generate a simple sine waveform.
asin oscil 20000, 440, 1

; Send the sine waveform to za variable #1.
zaw asin, 1
endin

instr 2 ;generates audio output.
; Read za variable #1.
a1 zar 1

; Generate audio output.
out a1

; Clear the za variables, get them ready for
; another pass.
zacl 0, 3
endin

instr 3 ;increments k-type channels
k0 zkr 0
k1 zkr 1
k2 zkr 2

zkw k0+1, 0
zkw k1+5, 1
zkw k2+10, 2
endin

instr 4 ;displays values from k-type channels
k0 zkr 0
k1 zkr 1
k2 zkr 2

; The total count for k0 is 30, since there are 10
; control blocks per second and instruments 3 and 4
; are on for 3 seconds.
printf "k0 = %i\n",k0, k0
printf "k1 = %i\n",k1, k1
printf "k2 = %i\n",k2, k2
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

i 1 0 1
i 2 0 1

i 3 0 3
i 4 0 3
e

</CsScore>
</CsoundSynthesizer>

```

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.45

Example written by Kevin Conder.

# zamod

zamod — Modulates one a-rate signal by a second one.

## Description

Modulates one a-rate signal by a second one.

## Syntax

```
ares zamod asig, kzamod
```

## Performance

*asig* -- the input signal

*kzamod* -- controls which za variable is used for modulation. A positive value means additive modulation, a negative value means multiplicative modulation. A value of 0 means no change to *asig*.

*zamod* modulates one a-rate signal by a second one, which comes from a za variable. The location of the modulating variable is controlled by the i-rate or k-rate variable *kzamod*. This is the a-rate version of *zkmod*.

## Examples

Here is an example of the zamod opcode. It uses the file *zamod.csd* [examples/zamod.csd].

### Example 1144. Example of the zamod opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o zamod.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 2 a-rate variables and 2 k-rate variables.
zakinit 2, 2

; Instrument #1 -- a simple waveform.
instr 1
; Vary an a-rate signal linearly from 20,000 to 0.
asig line 20000, p3, 0
```

```
    ; Send the signal to za variable #1.
    zaw asig, 1
  endin

  ; Instrument #2 -- generates audio output.
  instr 2
    ; Generate a simple sine wave.
    asin oscil 1, 440, 1

    ; Modify the sine wave, multiply its amplitude by
    ; za variable #1.
    a1 zamod asin, -1

    ; Generate the audio output.
    out a1

    ; Clear the za variables, prepare them for
    ; another pass.
    zacl 0, 2
  endin

</CsInstruments>
<CsScore>

  ; Table #1, a sine wave.
  f 1 0 16384 10 1

  ; Play Instrument #1 for 2 seconds.
  i 1 0 2
  ; Play Instrument #2 for 2 seconds.
  i 2 0 2
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*zacl, ziw, ziwM*

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.45

Example written by Kevin Conder.

# zar

zir — Reads from a location in za space at a-rate.

## Description

Reads from a location in za space at a-rate.

## Syntax

```
ares zar kndx
```

## Performance

*kndx* -- points to the za location to be read.

*zar* reads the array of floats at *kndx* in za space, which are ksmps number of a-rate floats to be processed in a k cycle.

## Examples

Here is an example of the zar opcode. It uses the file *zar.csd* [examples/zar.csd].

### Example 1145. Example of the zar opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o zar.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1

; Instrument #1 -- a simple waveform.
instr 1
; Generate a simple sine waveform.
asin oscil 20000, 440, 1

; Send the sine waveform to za variable #1.
zaw asin, 1
endin
```

```
; Instrument #2 -- generates audio output.
instr 2
  ; Read za variable #1.
  a1 zar 1

  ; Generate audio output.
  out a1

  ; Clear the za variables, get them ready for
  ; another pass.
  zacl 0, 1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #2 for one second.
i 2 0 1
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*zarg, zir, zkr*

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.45

Example written by Kevin Conder.

# zarg

zarg — Reads from a location in za space at a-rate, adds some gain.

## Description

Reads from a location in za space at a-rate, adds some gain.

## Syntax

```
ares zarg kndx, kgain
```

## Initialization

*kndx* -- points to the za location to be read.

*kgain* -- multiplier for the a-rate signal.

## Performance

*zarg* reads the array of floats at *kndx* in za space, which are ksmps number of a-rate floats to be processed in a k cycle. *zarg* also multiplies the a-rate signal by a k-rate value *kgain*.

## Examples

Here is an example of the zarg opcode. It uses the file *zarg.csd* [examples/zarg.csd].

### Example 1146. Example of the zarg opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o zarg.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1

; Instrument #1 -- a simple waveform.
instr 1
```

```
; Generate a simple sine waveform, with an amplitude
; between 0 and 1.
asin oscil 1, 440, 1

; Send the sine waveform to za variable #1.
zaw asin, 1
endin

; Instrument #2 -- generates audio output.
instr 2
; Read za variable #1, multiply its amplitude by 20,000.
a1 zarg 1, 20000

; Generate audio output.
out a1

; Clear the za variables, get them ready for
; another pass.
zACL 0, 1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #2 for one second.
i 2 0 1
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*zar, zir, zkr*

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.45

Example written by Kevin Conder.

## zaw

zaw — Writes to a za variable at a-rate without mixing.

## Description

Writes to a za variable at a-rate without mixing.

## Syntax

```
zaw asig, kndx
```

## Performance

*asig* -- value to be written to the za location.

*kndx* -- points to the zk or za location to which to write.

*zaw* writes *asig* into the za variable specified by *kndx*.

These opcodes are fast, and always check that the index is within the range of zk or za space. If not, an error is reported, 0 is returned, and no writing takes place.

## Examples

Here is an example of the *zaw* opcode. It uses the file *zaw.csd* [examples/zaw.csd].

### Example 1147. Example of the *zaw* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o zaw.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1

; Instrument #1 -- a simple waveform.
instr 1
; Generate a simple sine waveform.
```

```
asin oscil 2000, 440, 1

; Send the sine waveform to za variable #1.
zaw asin, 1
endin

; Instrument #2 -- generates audio output.
instr 2
; Read za variable #1.
a1 zar 1

; Generate the audio output.
out a1

; Clear the za variables, get them ready for
; another pass.
zacr 0, 1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #2 for one second.
i 2 0 1
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*zawm, ziw, ziwm, zkw, zkwm*

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.45

Example written by Kevin Conder.

## zawm

zawm — Writes to a za variable at a-rate with mixing.

## Description

Writes to a za variable at a-rate with mixing.

## Syntax

```
zawm asig, kndx [, imix]
```

## Initialization

*imix* (optional, default=1) -- indicates if mixing should occur.

## Performance

*asig* -- value to be written to the za location.

*kndx* -- points to the zk or za location to which to write.

These opcodes are fast, and always check that the index is within the range of zk or za space. If not, an error is reported, 0 is returned, and no writing takes place.

*zawm* is a mixing opcode, it adds the signal to the current value of the variable. If no *imix* is specified, mixing always occurs. *imix* = 0 will cause overwriting like *ziw*, *zkw*, and *zaw*. Any other value will cause mixing.

*Caution:* When using the mixing opcodes *ziwm*, *zkwm*, and *zawm*, care must be taken that the variables mixed to, are zeroed at the end (or start) of each k- or a-cycle. Continuing to add signals to them, can cause their values can drift to astronomical figures.

One approach would be to establish certain ranges of zk or za variables to be used for mixing, then use *zkcl* or *zacl* to clear those ranges.

## Examples

Here is an example of the *zawm* opcode. It uses the file *zawm.csd* [examples/zawm.csd].

### Example 1148. Example of the *zawm* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o zawm.wav -W ;; for file output any platform
```

```
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1

; Instrument #1 -- a basic instrument.
instr 1
; Generate a simple sine waveform.
asin oscil 15000, 440, 1

; Mix the sine waveform with za variable #1.
zawm asin, 1
endin

; Instrument #2 -- another basic instrument.
instr 2
; Generate another waveform with a different frequency.
asin oscil 15000, 880, 1

; Mix this sine waveform with za variable #1.
zawm asin, 1
endin

; Instrument #3 -- generates audio output.
instr 3
; Read za variable #1, containing both waveforms.
a1 zar 1

; Generate the audio output.
out a1

; Clear the za variables, get them ready for
; another pass.
zacl 0, 1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #2 for one second.
i 2 0 1
; Play Instrument #3 for one second.
i 3 0 1
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*zaw, ziw, ziwM, zkw, zkwm*

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.45

Example written by Kevin Conder.

# zdf\_1pole

`zdf_1pole` — Zero-delay feedback implementation of 1 pole filter.

## Description

Zero-delay feedback implementation of a 1 pole (6 dB/oct) filter. Offers low-pass (default), high-pass, and allpass output modes.

## Syntax

```
asig zdf_1pole ain, xcf [, kmode, istor]
```

## Initialization

`istor` --initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

`asig` -- high-pass output signal.

`asig` -- input signal.

`xcf` -- filter cutoff frequency (i-, k-, or a-rate).

`kmode` -- filter output type. The default value is 0. The output mode may be one of the following.

- 0 - low-pass
- 1 - high-pass
- 2 - allpass

## Examples

Here is an example of the `zdf_1pole` opcode. It uses the file `zdf_1pole.csd` [examples/zdf\_1pole.csd].

### Example 1149. Example of the `zdf_1pole` opcode.

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

sr = 48000
ksmps = 1
nchnls = 2
0dbfs = 1
```

```
instr 1
  asig = random:a(-1.0, 1.0)
  asig = zdf_lpole(asig, line(220, p3, 10000), p4)
  outc(asig, asig)
endin

instr 2
  asig = vco2(0.5, 220)
  asig = zdf_lpole(asig, line(220, p3, 10000), p4)
  outc(asig, asig)
endin

</CsInstruments>
<CsScore>
i1 0 4 0
i1 5 4 1
i1 10 4 2
i2 15 4 0
i2 20 4 1
i2 25 4 2
</CsScore>
</CsoundSynthesizer>
```

## References

This filter is based on the work of Will Pirkle that employs Vadim Zavalishin's work with bilinear transforms to create topology-preserving transform (TPT) implementations of analog filters.

1. Pirkle, Will. Designing Software Synthesizer Plug-ins in C++: For RackAFX, VST3, and Audio Units. CRC Press, 2014.
2. Pirkle, Will. AN-4: Virtual Analog (VA) Filter Implementation. 2013.
3. Zavalishin, Vadim. "The Art of VA filter design." Native Instruments, 2012.

## Credits

Author: Steven Yi  
April 2017

New in Csound 6.09.0

# zdf\_1pole\_mode

`zdf_1pole_mode` — Zero-delay feedback implementation of 1 pole filter with multimode output.

## Description

Zero-delay feedback implementation of a 1 pole (6 dB/oct) filter. Offers low-pass and high-pass output.

## Syntax

```
alp, ahp zdf_1pole_mode ain, xcf [, istor]
```

## Initialization

`istor` --initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

`alp` -- low-pass output signal.

`ahp` -- high-pass output signal.

`asig` -- input signal.

`xcf` -- filter cutoff frequency (i-, k-, or a-rate).

## Examples

Here is an example of the `zdf_1pole_mode` opcode. It uses the file `zdf_1pole_mode.csd` [examples/zdf\_1pole\_mode.csd].

### Example 1150. Example of the `zdf_1pole_mode` opcode.

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

sr = 48000
ksmps = 1
nchnls = 2
0dbfs = 1

instr 1
  asig = random:a(-1.0, 1.0)
  alp, ahp zdf_1pole_mode asig, line(220, p3, 10000)
  asig = (p4 == 0) ? alp : ahp
  outc(asig, asig)
endin
```

```
instr 2
  asig = vco2(0.5, 220)
  alp, ahp zdf_1pole_mode asig, line(220, p3, 10000)
  asig = (p4 == 0) ? alp : ahp
  outc(asig, asig)
endin

</CsInstruments>
<CsScore>
i1 0 4 0
i1 5 4 1
i2 10 4 0
i2 15 4 1
</CsScore>
</CsoundSynthesizer>
```

## References

This filter is based on the work of Will Pirkle that employs Vadim Zavalishin's work with bilinear transforms to create topology-preserving transform (TPT) implementations of analog filters.

1. Pirkle, Will. Designing Software Synthesizer Plug-ins in C++: For RackAFX, VST3, and Audio Units. CRC Press, 2014.
2. Pirkle, Will. AN-4: Virtual Analog (VA) Filter Implementation. 2013.
3. Zavalishin, Vadim. "The Art of VA filter design." Native Instruments, 2012.

## Credits

Author: Steven Yi  
April 2017

New in Csound 6.09.0

# zdf\_2pole

`zdf_2pole` — Zero-delay feedback implementation of 2 pole filter.

## Description

Zero-delay feedback implementation of a 2 pole (12 dB/oct) filter. Offers low-pass (default), high-pass, and allpass output modes.

## Syntax

```
asig zdf_2pole ain, xcf, xQ [, kmode, istor]
```

## Initialization

*istor* --initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*asig* -- output signal.

*ain* -- input signal.

*xcf* -- filter cutoff frequency (i-, k-, or a-rate).

*xQ* -- filter Q value (i-, k-, or a-rate). Range 0.5-25.0.

*kmode* -- filter output type. The default value is 0. The output mode may be one of the following.

- 0 - low-pass
- 1 - high-pass
- 2 - band-pass
- 3 - unity-gain band-pass
- 4 - notch
- 5 - allpass
- 6 - peak

## Examples

Here is an example of the `zdf_2pole` opcode. It uses the file `zdf_2pole.csd` [examples/zdf\_2pole.csd].

**Example 1151. Example of the `zdf_2pole` opcode.**

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 1
nchnls = 2
0dbfs = 1

instr 1
  asig = random:a(-1.0, 1.0)
  asig = zdf_2pole(asig, line(20, p3, 10000), 4, p4)

  outc(asig, asig)
endin

instr 2
  asig = vco2(0.5, 220)
  asig = zdf_2pole(asig, line(20, p3, 10000), 4, p4)
  outc(asig, asig)
endin

</CsInstruments>
<CsScore>
i1 0 4 0
i1 5 4 1
i1 10 4 2
i1 15 4 3
i1 20 4 4
i1 25 4 5
i1 30 4 6

i2 40 4 0
i2 45 4 1
i2 50 4 2
i2 55 4 3
i2 60 4 4
i2 65 4 5
i2 70 4 6

</CsScore>
</CsoundSynthesizer>
```

## References

This filter is based on the work of Will Pirkle that employs Vadim Zavalishin's work with bilinear transforms to create topology-preserving transform (TPT) implementations of analog filters.

1. Pirkle, Will. Designing Software Synthesizer Plug-ins in C++: For RackAFX, VST3, and Audio Units. CRC Press, 2014.
2. Pirkle, Will. AN-4: Virtual Analog (VA) Filter Implementation. 2013.
3. Zavalishin, Vadim. "The Art of VA filter design." Native Instruments, 2012.

## Credits

Author: Steven Yi  
April 2017

New in Csound 6.09.0

# zdf\_2pole\_mode

`zdf_2pole_mode` — Zero-delay feedback implementation of 2 pole filter with multimode output.

## Description

Zero-delay feedback implementation of a 2 pole (12 dB/oct) filter. Offers low-pass, band-pass, and high-pass output.

## Syntax

```
alp, abp, ahp zdf_2pole_mode ain, xcf, Q [, istor]
```

## Initialization

*istor* --initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

*alp* -- low-pass output signal.

*abp* -- band-pass output signal.

*ahp* -- high-pass output signal.

*ain* -- input signal.

*xcf* -- filter cutoff frequency (i-, k-, or a-rate).

*Q* -- filter Q value (i-, k-, or a-rate). Range 0.5-25.0.

## Examples

Here is an example of the `zdf_2pole_mode` opcode. It uses the file `zdf_2pole_mode.csd` [examples/zdf\_2pole\_mode.csd].

### Example 1152. Example of the `zdf_2pole_mode` opcode.

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 1
nchnls = 2
0dbfs = 1

instr 1
```

```
asig = random:a(-1.0, 1.0)
asigs[] init 3
asigs[0], asigs[1], asigs[2] zdf_2pole_mode asig, line(20, p3, 10000), 4

asig = asigs[p4]
outc(asig, asig)
endin

instr 2
asig = vco2(0.5, 220)
asigs[] init 3
asigs[0], asigs[1], asigs[2] zdf_2pole_mode asig, line(20, p3, 10000), 4
outc(asigs[p4], asigs[p4])
endin

</CsInstruments>
<CsScore>
i1 0 4 0
i1 5 4 1
i1 10 4 2

i2 15 4 0
i2 20 4 1
i2 25 4 2

</CsScore>
</CsoundSynthesizer>
```

## References

This filter is based on the work of Will Pirkle that employs Vadim Zavalishin's work with bilinear transforms to create topology-preserving transform (TPT) implementations of analog filters.

1. Pirkle, Will. Designing Software Synthesizer Plug-ins in C++: For RackAFX, VST3, and Audio Units. CRC Press, 2014.
2. Pirkle, Will. AN-4: Virtual Analog (VA) Filter Implementation. 2013.
3. Zavalishin, Vadim. "The Art of VA filter design." Native Instruments, 2012.

## Credits

Author: Steven Yi  
April 2017

New in Csound 6.09.0

# zdf\_ladder

`zdf_ladder` — Zero-delay feedback implementation of 4 pole ladder filter.

## Description

Zero-delay feedback implementation of a 4 pole (24 dB/oct) low-pass filter based on the Moog ladder filter.

## Syntax

```
asig zdf_ladder ain, xcf, xQ [, istor]
```

## Initialization

`istor` --initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

## Performance

`asig` -- output signal.

`ain` -- input signal.

`xcf` -- filter cutoff frequency (i-, k-, or a-rate).

`xQ` -- filter Q value (i-, k-, or a-rate). Range 0.5-25.0. Self-oscillation occurs at 25.0.

## Examples

Here is an example of the `zdf_ladder` opcode. It uses the file `zdf_ladder.csd` [examples/zdf\_ladder.csd].

### Example 1153. Example of the `zdf_ladder` opcode.

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

sr = 48000
ksmps = 1
nchnls = 2
0dbfs = 1

instr 1
  asig = random:a(-1.0, 1.0)
  asig = zdf_ladder(asig, expon(220, p3, 5000), p4)
  outc(asig, asig)
endin

instr 2
  asig = vco2(0.5, 220)
```

```

    asig = zdf_ladder(asig, expon(220, p3, 5000), p4)
    outc(asig, asig)
  endin

instr 3
  asig = vco2(0.5, p4)
  asig = zdf_ladder(asig, expon(5000, p3, 200), 0.5 + p5* 24)
  asig *= madsr(0.05, 0, 1, 0.25)
  outc(asig, asig)
endin

instr play_instr3
  schedule(3, 0, 0.25, mtof(48 + (p4 % 2) * 12), p4 / 16)

  if(p4 < 16) then
    schedule("play_instr3", 0.25, 0.25, p4 + 1)
  else
    event_i("e", 0.5, 0)
  endif
  turnoff
endin

</CsInstruments>
<CsScore>
i1 0 2 0.5
i1 + . 1
i1 + . 4
i1 + . 10
i1 + . 18
i1 + . 24.5
i2 12 2 0.5
i2 + . 1
i2 + . 4
i2 + . 10
i2 + . 18
i2 + . 24.5

s
i "play_instr3" 0 0.25 0
f0 60
</CsScore>
</CsoundSynthesizer>

```

## References

This filter is based on the work of Will Pirkle that employs Vadim Zavalishin's work with bilinear transforms to create topology-preserving transform (TPT) implementations of analog filters.

1. Pirkle, Will. Designing Software Synthesizer Plug-ins in C++: For RackAFX, VST3, and Audio Units. CRC Press, 2014.
2. Pirkle, Will. AN-4: Virtual Analog (VA) Filter Implementation. 2013.
3. Zavalishin, Vadim. "The Art of VA filter design." Native Instruments, 2012.

## Credits

Author: Steven Yi  
April 2017

New in Csound 6.09.0

## zfilter2

`zfilter2` — Performs filtering using a transposed form-II digital filter lattice with radial pole-shearing and angular pole-warping.

### Description

General purpose custom filter with time-varying pole control. The filter coefficients implement the following difference equation:

$$(1)*y(n) = b0*x[n] + b1*x[n-1] + \dots + bM*x[n-M] - a1*y[n-1] - \dots - aN*y[n-N]$$

the system function for which is represented by:

$$H(Z) = \frac{B(Z)}{A(Z)} = \frac{b0 + b1*Z^{-1} + \dots + bM*Z^{-M}}{1 + a1*Z^{-1} + \dots + aN*Z^{-N}}$$

### Syntax

```
ares zfilter2 asig, kdamp, kfreq, iM, iN, ib0, ib1, ..., ibM, \  
      ia1, ia2, ..., iaN
```

### Initialization

At initialization the number of zeros and poles of the filter are specified along with the corresponding zero and pole coefficients. The coefficients must be obtained by an external filter-design application such as Matlab and specified directly or loaded into a table via *GEN01*. With *zfilter2*, the roots of the characteristic polynomials are solved at initialization so that the pole-control operations can be implemented efficiently.

### Performance

The *filter2* opcodes perform filtering using a transposed form-II digital filter lattice with no time-varying control. *zfilter2* uses the additional operations of radial pole-shearing and angular pole-warping in the Z plane.

Pole shearing increases the magnitude of poles along radial lines in the Z-plane. This has the affect of altering filter ring times. The k-rate variable *kdamp* is the damping parameter. Positive values (0.01 to 0.99) increase the ring-time of the filter (hi-Q), negative values (-0.01 to -0.99) decrease the ring-time of the filter, (lo-Q).

Pole warping changes the frequency of poles by moving them along angular paths in the Z plane. This operation leaves the shape of the magnitude response unchanged but alters the frequencies by a constant factor (preserving 0 and p). The k-rate variable *kfreq* determines the frequency warp factor. Positive values (0.01 to 0.99) increase frequencies toward p and negative values (-0.01 to -0.99) decrease frequencies toward 0.

Since *filter2* implements generalized recursive filters, it can be used to specify a large range of general DSP algorithms. For example, a digital waveguide can be implemented for musical instrument modeling using a pair of *delayr* and *delayw* opcodes in conjunction with the *filter2* opcode.

## Examples

A controllable second-order IIR filter operating on an a-rate signal:

```
a1 zfilter2 asig, kdamp, kfreq, 1, 2, 1, ia1, ia2 ;; controllable a-rate IIR filter
```

## See Also

*filter2*

## Credits

Author: Michael A. Casey  
M.I.T.  
Cambridge, Mass.  
1997

New in Version 3.47

# zir

zir — Reads from a location in zk space at i-rate.

## Description

Reads from a location in zk space at i-rate.

## Syntax

```
ir zir indx
```

## Initialization

*indx* -- points to the zk location to be read.

## Performance

*zir* reads the signal at *indx* location in zk space.

## Examples

Here is an example of the zir opcode. It uses the file *zir.csd* [examples/zir.csd].

### Example 1154. Example of the zir opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o zir.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1

; Instrument #1 -- a simple instrument.
instr 1
; Set the zk variable #1 to 32.594.
ziw 32.594, 1
endin
```

```
; Instrument #2 -- prints out zk variable #1.
instr 2
  ; Read the zk variable #1 at i-rate.
  i1 zir 1

  ; Print out the value of zk variable #1.
  print i1
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #2 for one second.
i 2 0 1
e

</CsScore>
</CsSoundSynthesizer>
```

## See Also

*zar, zarg, zkr*

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.45

Example written by Kevin Conder.

# ziw

ziw — Writes to a zk variable at i-rate without mixing.

## Description

Writes to a zk variable at i-rate without mixing.

## Syntax

```
ziw isig, indx
```

## Initialization

*isig* -- initializes the value of the zk location.

*indx* -- points to the zk or za location to which to write.

## Performance

*ziw* writes *isig* into the zk variable specified by *indx*.

These opcodes are fast, and always check that the index is within the range of zk or za space. If not, an error is reported, 0 is returned, and no writing takes place.

## Examples

Here is an example of the *ziw* opcode. It uses the file *ziw.csd* [examples/ziw.csd].

### Example 1155. Example of the *ziw* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o ziw.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1
```

```
; Instrument #1 -- a simple instrument.
instr 1
  ; Set zk variable #1 to 64.182.
  ziw 64.182, 1
endin

; Instrument #2 -- prints out zk variable #1.
instr 2
  ; Read zk variable #1 at i-rate.
  i1 zir 1

  ; Print out the value of zk variable #1.
  print i1
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #2 for one second.
i 2 0 1
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*zaw, zawm, ziwm, zkw, zkwm*

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.45

Example written by Kevin Conder.

# ziwm

ziwm — Writes to a zk variable to an i-rate variable with mixing.

## Description

Writes to a zk variable to an i-rate variable with mixing.

## Syntax

```
ziwm isig, indx [, imix]
```

## Initialization

*isig* -- initializes the value of the zk location.

*indx* -- points to the zk location location to which to write.

*imix* (optional, default=1) -- indicates if mixing should occur.

## Performance

*ziwm* is a mixing opcode, it adds the signal to the current value of the variable. If no *imix* is specified, mixing always occurs. *imix* = 0 will cause overwriting like *ziw*, *zkw*, and *zaw*. Any other value will cause mixing.

*Caution:* When using the mixing opcodes *ziwm*, *zkwm*, and *zawm*, care must be taken that the variables mixed to, are zeroed at the end (or start) of each k- or a-cycle. Continuing to add signals to them, can cause their values can drift to astronomical figures.

One approach would be to establish certain ranges of zk or za variables to be used for mixing, then use *zkcl* or *zacl* to clear those ranges.

## Examples

Here is an example of the *ziwm* opcode. It uses the file *ziwm.csd* [examples/ziwm.csd].

### Example 1156. Example of the *ziwm* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o ziwm.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1

; Instrument #1 -- a simple instrument.
instr 1
  ; Add 20.5 to zk variable #1.
  ziwm 20.5, 1
endin

; Instrument #2 -- another simple instrument.
instr 2
  ; Add 15.25 to zk variable #1.
  ziwm 15.25, 1
endin

; Instrument #3 -- prints out zk variable #1.
instr 3
  ; Read zk variable #1 at i-rate.
  i1 zir 1

  ; Print out the value of zk variable #1.
  ; It should be 35.75 (20.5 + 15.25)
  print i1
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #2 for one second.
i 2 0 1
; Play Instrument #3 for one second.
i 3 0 1
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*zaw, zawm, ziw, zkw, zkwm*

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.45

Example written by Kevin Conder.

# zkcl

zkcl — Clears one or more variables in the zk space.

## Description

Clears one or more variables in the zk space.

## Syntax

```
zkcl kfirst, klast
```

## Performance

*ksig* -- the input signal

*kfirst* -- first zk or za location in the range to clear.

*klast* -- last zk or za location in the range to clear.

*zkcl* clears one or more variables in the zk space. This is useful for those variables which are used as accumulators for mixing k-rate signals at each cycle, but which must be cleared before the next set of calculations.

## Examples

Here is an example of the *zkcl* opcode. It uses the file *zkcl.csd* [examples/*zkcl.csd*].

### Example 1157. Example of the *zkcl* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac        -iadc      -d          ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o zkcl.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1

; Instrument #1 -- a simple waveform.
instr 1
```

```
; Linearly vary a k-rate signal from 220 to 1760.
kline line 220, p3, 1760

; Add the linear signal to zk variable #1.
zkw kline, 1
endin

; Instrument #2 -- generates audio output.
instr 2
; Read zk variable #1.
kfreq zkr 1

; Use the value of zk variable #1 to vary
; the frequency of a sine waveform.
a1 oscil 20000, kfreq, 1

; Generate the audio output.
out a1

; Clear the zk variables, get them ready for
; another pass.
zkcl 0, 1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for three seconds.
i 1 0 3
; Play Instrument #2 for three seconds.
i 2 0 3
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*zacl, zkwm, zkw, zkmod, zkr*

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.45

Example written by Kevin Conder.

# zkmod

zkmod — Facilitates the modulation of one signal by another.

## Description

Facilitates the modulation of one signal by another.

## Syntax

```
kres zkmod ksig, kzkmod
```

## Performance

*ksig* -- the input signal

*kzkmod* -- controls which zk variable is used for modulation. A positive value means additive modulation, a negative value means multiplicative modulation. A value of 0 means no change to *ksig*. *kzkmod* can be i-rate or k-rate

*zkmod* facilitates the modulation of one signal by another, where the modulating signal comes from a zk variable. Either additive or multiplicative modulation can be specified.

## Examples

Here is an example of the zkmod opcode. It uses the file *zkmod.csd* [examples/zkmod.csd].

### Example 1158. Example of the zkmod opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o zkmod.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

; Initialize the ZAK space.
; Create 2 a-rate variables and 2 k-rate variables.
zakinit 2, 2

; Instrument #1 -- a signal with jitter.
instr 1
; Generate a k-rate signal goes from 30 to 2,000.
```

```
kline line 30, p3, 2000

; Add the signal into zk variable #1.
zkw kline, 1
endin

; Instrument #2 -- generates audio output.
instr 2
; Create a k-rate signal modulated the jitter opcode.
kamp init 20
kcpsmin init 40
kcpsmax init 60
kjtr jitter kamp, kcpsmin, kcpsmax

; Get the frequency values from zk variable #1.
kfreq zkr 1
; Add the the frequency values in zk variable #1 to
; the jitter signal.
kjfreq zkmod kjtr, 1

; Use a simple sine waveform for the left speaker.
aleft oscil 20000, kfreq, 1
; Use a sine waveform with jitter for the right speaker.
aright oscil 20000, kjfreq, 1

; Generate the audio output.
outs aleft, aright

; Clear the zk variables, prepare them for
; another pass.
zkcl 0, 2
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for 2 seconds.
i 1 0 2
; Play Instrument #2 for 2 seconds.
i 2 0 2
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*zamod, zkcl, zkr, zkwm, zkw*

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.45

Example written by Kevin Conder.

# zkr

zkr — Reads from a location in zk space at k-rate.

## Description

Reads from a location in zk space at k-rate.

## Syntax

```
kres zkr kndx
```

## Initialization

*kndx* -- points to the zk location to be read.

## Performance

*zkr* reads the array of floats at *kndx* in zk space.

## Examples

Here is an example of the zkr opcode. It uses the file *zkr.csd* [examples/zkr.csd].

### Example 1159. Example of the zkr opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d          ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o zkr.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1

; Instrument #1 -- a simple waveform.
instr 1
; Linearly vary a k-rate signal from 440 to 880.
kline line 440, p3, 880

; Add the linear signal to zk variable #1.
```

```
    zkw kline, 1
  endin

  ; Instrument #2 -- generates audio output.
  instr 2
    ; Read zk variable #1.
    kfreq zkr 1

    ; Use the value of zk variable #1 to vary
    ; the frequency of a sine waveform.
    a1 oscil 20000, kfreq, 1

    ; Generate the audio output.
    out a1

    ; Clear the zk variables, get them ready for
    ; another pass.
    zkcl 0, 1
  endin

</CsInstruments>
<CsScore>

  ; Table #1, a sine wave.
  f 1 0 16384 10 1

  ; Play Instrument #1 for one second.
  i 1 0 1
  ; Play Instrument #2 for one second.
  i 2 0 1
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*zar, zarg, zir, zkcl, zkmod, zkwm, zkw*

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.45

Example written by Kevin Conder.

# zkw

zkw — Writes to a zk variable at k-rate without mixing.

## Description

Writes to a zk variable at k-rate without mixing.

## Syntax

```
zkw ksig, kndx
```

## Performance

*ksig* -- value to be written to the zk location.

*kndx* -- points to the zk or za location to which to write.

*zkw* writes *ksig* into the zk variable specified by *kndx*.

## Examples

Here is an example of the zkw opcode. It uses the file *zkw.csd* [examples/zkw.csd].

### Example 1160. Example of the zkw opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o zkw.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1

; Instrument #1 -- a simple waveform.
instr 1
; Linearly vary a k-rate signal from 100 to 1,000.
kline line 100, p3, 1000

; Add the linear signal to zk variable #1.
zkw kline, 1
```

```
endin

; Instrument #2 -- generates audio output.
instr 2
  ; Read zk variable #1.
  kfreq zkr 1

  ; Use the value of zk variable #1 to vary
  ; the frequency of a sine waveform.
  a1 oscil 20000, kfreq, 1

  ; Generate the audio output.
  out a1

  ; Clear the zk variables, get them ready for
  ; another pass.
  zkcl 0, 1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
; Play Instrument #2 for two seconds.
i 2 0 2
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*zaw, zawm, ziw, ziwm, zkr, zkwm*

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.45

Example written by Kevin Conder.

## zkwm

zkwm — Writes to a zk variable at k-rate with mixing.

## Description

Writes to a zk variable at k-rate with mixing.

## Syntax

```
zkwm ksig, kndx [, imix]
```

## Initialization

*imix* (optional, default=1) -- indicates if mixing should occur.

## Performance

*ksig* -- value to be written to the zk location.

*kndx* -- points to the zk or za location to which to write.

*zkwm* is a mixing opcode, it adds the signal to the current value of the variable. If no *imix* is specified, mixing always occurs. *imix* = 0 will cause overwriting like *ziw*, *zkw*, and *zaw*. Any other value will cause mixing.

*Caution:* When using the mixing opcodes *ziwm*, *zkwm*, and *zawm*, care must be taken that the variables mixed to, are zeroed at the end (or start) of each k- or a-cycle. Continuing to add signals to them, can cause their values can drift to astronomical figures.

One approach would be to establish certain ranges of zk or za variables to be used for mixing, then use *zkl* or *zawl* to clear those ranges.

## Examples

Here is an example of the *zkwm* opcode. It uses the file *zkwm.csd* [examples/zkwm.csd].

### Example 1161. Example of the *zkwm* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d      ;;RT audio I/O
; For Non-realtime output leave only the line below:
; -o zkwm.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
```

```
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1

; Instrument #1 -- a basic instrument.
instr 1
  ; Generate a k-rate signal.
  ; The signal goes from 30 to 20,000 then back to 30.
  kramp linseg 30, p3/2, 20000, p3/2, 30

  ; Mix the signal into the zk variable #1.
  zkwm kramp, 1
endin

; Instrument #2 -- another basic instrument.
instr 2
  ; Generate another k-rate signal.
  ; This is a low frequency oscillator.
  klfo lfo 3500, 2

  ; Mix this signal into the zk variable #1.
  zkwm klfo, 1
endin

; Instrument #3 -- generates audio output.
instr 3
  ; Read zk variable #1, containing a mix of both signals.
  kamp zkr 1

  ; Create a sine waveform. Its amplitude will vary
  ; according to the values in zk variable #1.
  a1 oscil kamp, 880, 1

  ; Generate the audio output.
  out a1

  ; Clear the zk variable, get it ready for
  ; another pass.
  zkcl 0, 1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for 5 seconds.
i 1 0 5
; Play Instrument #2 for 5 seconds.
i 2 0 5
; Play Instrument #3 for 5 seconds.
i 3 0 5
e

</CsScore>
</CsSoundSynthesizer>
```

## See Also

*zaw, zawm, ziw, ziwm, zkcl, zkw, zkr*

## Credits

Author: Robin Whittle  
Australia  
May 1997

New in version 3.45

Example written by Kevin Conder.

---

# Score Statements and GEN Routines

## Score Statements

The statements used in scores are:

- *a* - Advance score time by a specified amount
- *b* - Resets the clock
- *C* - Control automatic carry of p-args
- *e* - Marks the end of the last section of the score
- *f* - Causes a *GEN subroutine* to place values in a stored function table
- *i* - Makes an instrument active at a specific time and for a certain duration
- *m* - Sets a named mark in the score
- *n* - Repeats a marked section
- *q* - Used to quiet an instrument
- *r* - Starts a repeated section
- *s* - Marks the end of a section
- *t* - Sets the tempo
- *v* - Provides for locally variable time warping of score events
- *x* - Skip the rest of the current section
- *y* - Set seed for random numbers, either from p1 or, if omitted, the clock
- *{* - Begins a non-sectional, nestable loop
- *}* - Ends a non-sectional, nestable loop

# a Statement (or Advance Statement)

a — Advance score time by a specified amount.

## Description

This causes score time to be advanced by a specified amount without producing sound samples.

## Syntax

a p1 p2 p3

## Performance

p1 Carries no meaning. Usually zero.  
 p2 Action time, in beats, at which advance is to begin.  
 p3 Number of beats to advance without producing sound.  
 p4 |  
 p5 | These carry no meaning.  
 p6 |  
 .  
 .

## Special Considerations

This statement allows the beat count within a score section to be advanced without generating intervening sound samples. This can be of use when a score section is incomplete (the beginning or middle is missing) and the user does not wish to generate and listen to a lot of silence.

p2, action time, and p3, number of beats, are treated as in *i statements*, with respect to sorting and modification by *t statements*.

An *a statement* will be temporarily inserted in the score by the Score Extract feature when the extracted segment begins later than the start of a Section. The purpose of this is to preserve the beat count and time count of the original score for the benefit of the peak amplitude messages which are reported on the user console.

Whenever an *a statement* is encountered by a performing orchestra, its presence and effect will be reported on the user's console.

## Examples

Here is an example of the a statement. It uses the file *a.csd* [examples/a.csd].

### Example 1162. Example of the a statement.

```
<CsoundSynthesizer>
<CsOptions>
```

```
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o a.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

aenv expseg .01, p3*0.25, 1, p3*0.75, 0.01
asig poscil3 .8*aenv, p4
      outs asig, asig

endin
</CsInstruments>
<CsScore>
;two sections
s
a 0 0 6 ;advance score 6 seconds
i 1 0 2 110 ;these first 2 notes
i 1 3 2 220 ;will not sound
i 1 6 2 440
i 1 9 2 880
s
a 0 3 6 ;advance score 6 seconds, but do this after 3 seconds
i 1 0 2 110 ;this will sound, because action time (p2) from a statement = 3
i 1 3 2 220 ;so these 2 notes
i 1 6 2 440 ;will not sound
i 1 9 2 880 ;and this one will
e
</CsScore>
</CsoundSynthesizer>
```

## b Statement

b Statement — This statement resets the clock.

### Description

This statement resets the clock.

### Syntax

`b p1`

### Performance

*p1* -- Specifies how the clock is to be set.

### Special Considerations

*p1* is the number of beats by which *p2* values of subsequent *i statements* are modified. If *p1* is positive, the clock is reset forward, and subsequent notes appear later, the number of beats specified by *p1* being added to the note's *p2*. If *p1* is negative, the clock is reset backward, and subsequent notes appear earlier, the number of beats specified by *p1* being subtracted from the note's *p2*. There is no cumulative affect. The clock is reset with each *b statement*. If *p1* = 0, the clock is returned to its original position, and subsequent notes appear at their specified *p2*.

### Examples

Here is an example of the *b* statement. It uses the file *b.csd* [examples/b.csd].

#### Example 1163. Example of the *b* statement.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o b.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

aenv expseg .01, p3*0.25, 1, p3*0.75, 0.01
asig poscil3 .4*aenv, 220, 1
outs asig, asig

endin
```

```
instr 2
asig pluck 0.7, p4, 220, 0, 1
outs asig, asig

endin

instr 3
asig loscil .8, 1, 2, 1
outs asig, asig

endin

instr 4
asig bamboo .8, 0.01
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine wave
f 2 0 0 1 "fox.wav" 0 0 0 ;sample

i1 0 2
i1 10 2

b 5 ; set the clock "forward"
i2 1 2 220 ; start time = 6
i2 2 2 110 ; start time = 7

b -1
i3 3 2 ; start time = 2
i3 5.5 1 ; start time = 4.5

b 0 ; reset clock to normal
i4 10 2 ; start time = 10

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Explanation suggested and example provided by Paul Winkler. (Csound Version 4.07)

# C Statement

C Statement — This statement switches automatic carrying of arguments on or off.

## Description

This statement switches automatic carrying of missing arguments on or off.

## Syntax

`C pl`

## Performance

`pl` -- Specifies whether to switch off (value 0) or on (non-zero).

## Examples

Here is an example of the C statement. It uses the file *C.csd* [examples/b.csd].

### Example 1164. Example of the C statement.

```
<CsoundSynthesizer>

<CsInstruments>
Odbfs = 1

instr 1
i i pcount
print i i
  a1 oscil 0.5,A4
  outs a1,a1
endin
</CsInstruments>

<CsScore>

i1 0 .5 100
i . +
i . . . !
i

s
i1 0 .5 100
i . + .
C 0
i . . .
i . .
i .

e
</CsScore>

</CsoundSynthesizer>
```

## Credits

Author: John ffitth  
Alta Sounds  
Bath, UK  
2016

New in version 6.08

# d Statement (De Note Statement)

`d` — removes an active instrument at a specific time.

## Description

This statement calls for an instrument that has been held to be removed at a specific time.

## Syntax

```
i p1 p2 p3 p4
```

## Initialization

*p1* -- Instrument number. An optional fractional part can provide an additional tag for specifying ties between particular notes of consecutive clusters.

*p2* -- Starting time in arbitrary units called beats.

*p3* -- ignored but required. Usually zero.

## Performance

Beats are evaluated as seconds, unless there is a *t statement* in this score section or a *-t flag* in the command-line.

Action times are relative to the beginning of a section ( see *s statement*), which is assigned time 0.

Note statements within a section may be placed in any order. Before being sent to an orchestra, unordered score statements must first be processed by Sorter, which will reorder them by ascending *p2* value. Notes with the same *p2* value will be ordered by ascending *p1*; if the same *p1*, then by ascending *p3*.

## Examples

Here is an example of the `d` statement. It uses the file `d_statement.csd` [examples/d\_statement.csd].

### Example 1165. Example of the `d` statement.

```
<CsoundSynthesizer>

<CsInstruments>
instr 1
  a1 oscil 10000, 440
    out a1
endin
instr sound
  a1 oscil 10000, 440
    out a1
endin
</CsInstruments>

<CsScore>
```

```
f 0 10
i 1 0 -1
d 1 1 0

i "sound" 3 -1
i "-sound" 4 0
e
</CsScore>

</CsoundSynthesizer>
```

## Credits

Author: John ffitch, new in version 6.09.

# e Statement

*e* statement — This statement may be used to mark the end of the last section of the score.

## Description

This statement may be used to mark the end of the last section of the score.

## Syntax

```
e [0 time]
```

## Performance

The second p-field *time* is optional and if present determines the end time (length in beats) of the final section of the score. This time must be after the end of the last event otherwise it will have no effect. "Always on" instruments will end at the given time. Extending the section in this way is useful to avoid prematurely cutting off reverb tails or other effects.

## Special Considerations

The *e* statement is contextually identical to an *s* statement. Additionally, the *e* statement terminates all signal generation (including indefinite performance) and closes all input and output files.

If an *e* statement occurs before the end of a score, all subsequent score lines will be ignored.

The *e* statement is optional in a score file yet to be sorted. If a score file has no *e* statement, then Sort processing will supply one.

## Examples

Here is an example of the *e* statement. It uses the file *e.csd* [examples/e.csd].

### Example 1166. Example of the *e* statement.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o e.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gal init 0

instr 1
```

```
aenv expseg .01, p3*0.1, 1, p3*0.9, 0.01
gal poscil3 .5*aenv, cpspch(p4), 1
    outs gal,gal
endin

instr 99

aL, aR reverbsc gal, gal, 0.85, 12000, sr, 0.5, 1
outs aL,aR

gal = 0

endin
</CsInstruments>
<CsScore>
f 1 0 128 10 1 ;sine wave

i 1 1 0.1 8.00
i 1 2 0.1 8.02
i 1 3 0.1 8.04
i 1 4 0.1 9.06

i 99 0 6 ;remains active for 6 seconds

e10
</CsScore>
</CsoundSynthesizer>
```

# f Statement (or Function Table Statement)

f Statement (or Function Table Statement) — Causes a GEN subroutine to place values in a stored function table.

## Description

This causes a GEN subroutine to place values in a stored function table for use by instruments.

## Syntax

```
f p1 p2 p3 p4 p5 ... PMAX
```

## Performance

*p1* -- Table number by which the stored function will be known. A negative number requests that the table be destroyed.

*p2* -- Action time of function generation (or destruction) in beats.

*p3* -- Size of function table (i.e. number of points) Must be a power of 2, or a power-of-2 plus 1 if this number is positive. Maximum table size is 16777216 ( $2^{24}$ ) points.

*p4* -- Number of the GEN routine to be called (see *GEN ROUTINES*). A negative value will cause rescaling to be omitted.

*p5 ... PMAX* -- Parameters whose meaning is determined by the particular GEN routine.

## Special Considerations

Function tables are arrays of floating-point values. You can create a simple sine wave using the line:

```
f 1 0 1024 10 1
```

This table uses *GEN10* to fill the table.

Historically, due to older platform constraints, Csound could only accept tables whose size was a power of two. This limitation has been removed in recent versions, and you can freely create tables of any size. However, to create a table whose size is not a power of two (or power of two plus one), you must specify the size as a negative number.



### Note

Not all opcodes accept tables whose size is not a power of two, since they may depend on this for internal optimization.

For arrays whose length is a power of 2, space allocation always provides for  $2^n$  points plus an additional *guard point*. The guard point value, used during interpolated lookup, can be automatically set to reflect the table's purpose: If *size* is an exact power of 2, the guard point will be a copy of the first point; this is appropriate for *interpolated wrap-around lookup* as in *oscili*, etc., and should even be used for non-interpolating *oscil* for safe consistency. If *size* is set to  $2^n + 1$ , the guard point value automatically extends the contour of table values; this is appropriate for single-scan functions such in *envplx*, *oscil1*, *oscilli*, etc.

The size of the table is used as a code to tell Csound how to fill this guard-point. If the size is exactly power-of-two, then the guard point contains a copy of the first point on the table. If the size is power-of-two plus one, Csound will extend the contour of the function stored in the table for one extra point.

Table space is allocated in primary memory, along with instrument data space. The maximum table number used to be 200. This has been changed to be limited by memory only. (Currently there is an internal soft limit of 300, this is automatically extended as required.)

An existing function table can be removed by an *f statement* containing a negative p1 and an appropriate action time. A function table can also be removed by the generation of another table with the same p1. Functions are not automatically erased at the end of a score section.

p2 action time is treated in the same way as in *i statements* with respect to sorting and modification by *t statements*. If an *f statement* and an *i statement* have the same p2, the sorter gives the *f statement* precedence so that the function table will be available during note initialization.



### Warning

The maximum number of p-fields accepted in the score is determined by PMAX (a compilation time variable). PMAX is currently set to 1000. This may discard values entered when using *GEN02*. To overcome this, use *GEN23* or *GEN28* to read the values from a file.

An *f 0 statement* (zero p1, positive p2) may be used to create an action time with no associated action. Such time markers are useful for padding out a score section (see *s statement*) and for letting Csound run from realtime events only (e.g. using only MIDI input without score events). The time given is the number of seconds Csound will run. If you want Csound to run for 10 hours, use:

```
f0 36000
```

The simplest way to fill a table (f1) with 0's is:

```
f1 0 xx 2 0
```

where xx = table size.

The simplest way to fill a table (f1) with \*any\* single value is:

```
f1 0 xx -7 yy xx yy
```

where xx = table size and yy = any single value

In both of the above examples, table size (p3) must be a power of 2 or power-of-2 plus 1.

## See also

*GEN ROUTINES*

## Credits

Updated August 2002 thanks to a note from Rasmus Ekman. There is no longer a hard limit of 200 function tables.

# i Statement (Instrument or Note Statement)

*i* — Makes an instrument active at a specific time and for a certain duration.

## Description

This statement calls for an instrument to be made active at a specific time and for a certain duration. The parameter field values are passed to that instrument prior to its initialization, and remain valid throughout its Performance.

## Syntax

```
i p1 p2 p3 p4 ...
```

## Initialization

*p1* -- Instrument number, usually a non-negative integer. An optional fractional part can provide an additional tag for specifying ties between particular notes of consecutive clusters. A negative *p1* (including tag) can be used to turn off a particular “held” note. In the case of a named instrument it can be negated by having a '-' as the first character of the string.

*p2* -- Starting time in arbitrary units called beats.

*p3* -- Duration time in beats (usually positive). A negative value will initiate a held note (see also *ihold*). A negative value can also be used for 'always on' instruments like reverberation. These notes are not terminated by *s statements*. A zero value will invoke an initialization pass without performance (see also *instr*).

*p4* ... -- Parameters whose significance is determined by the instrument.

## Performance

Beats are evaluated as seconds, unless there is a *t statement* in this score section or a *-t flag* in the command-line.

Starting or action times are relative to the beginning of a section ( see *s statement*), which is assigned time 0.

Note statements within a section may be placed in any order. Before being sent to an orchestra, unordered score statements must first be processed by Sorter, which will reorder them by ascending *p2* value. Notes with the same *p2* value will be ordered by ascending *p1*; if the same *p1*, then by ascending *p3*.

Notes may be stacked, i.e., a single instrument can perform any number of notes simultaneously. (The necessary copies of the instrument's data space will be allocated dynamically by the orchestra loader.) Each note will normally turn off when its *p3* duration has expired, or on receipt of a MIDI noteoff signal. An instrument can modify its own duration either by changing its *p3* value during note initialization, or by prolonging itself through the action of a *linenr* or *xrtrim* unit.

An instrument may be turned on and left to perform indefinitely either by giving it a negative *p3* or by including an *ihold* in its *i-time* code. If a held note is active, an *i statement with matching p1* will not cause a new allocation but will take over the data space of the held note. The new *pfields* (including *p3*) will now be in effect, and an *i-time* pass will be executed in which the units can either be newly initialized or allowed to continue as required for a tied note (see *tigoto*). A held note may be succeeded either by another

held note or by a note of finite duration. A held note will continue to perform across section endings (see *s statement*). It is halted only by *turnoff* or by an *i statement* with negative matching p1 or by an *e statement*.

It is possible to have multiple instances (usually, but not necessarily, notes of different pitches) of the same instrument, held simultaneously, via negative p3 values. The instrument can then be fed new parameters from the score. This is useful for avoiding long hard-coded *linsegs*, and can be accomplished by adding a decimal part to the instrument number.

For example, to hold three copies of instrument 10 in a simple chord:

```
i10.1  0  -1  7.00
i10.2  0  -1  7.04
i10.3  0  -1  7.07
```

Subsequent *i* statements can refer to the same sounding note instances, and if the instrument definition is done properly, the new p-fields can be used to alter the character of the notes in progress. For example, to bend the previous chord up an octave and release it:

```
i10.1  1  1  8.00
i10.2  1  1  8.04
i10.3  1  1  8.07
```



### Tip

When turning off notes, bear in mind that  $i\ 1.1 == i\ 1.10$  and  $i\ 1.1 != i\ 1.01$ . The maximum number of decimal places that can be used depends on the precision Csound was compiled with (See *Csound Double (64-bit) vs. Float (32-bit)*)

The instrument definition has to take this into account, however, especially if clicks are to be avoided (see the example below).

Note that the decimal instrument number notation cannot be used in conjunction with real-time MIDI. In this case, the instrument would be monophonic while a note was held.

Notes being tied to previous instances of the same instrument, should skip most initialization by means of *tigoto*, except for the values entered in score. For example, all table reading opcodes in the instrument, should usually be skipped, as they store their phase internally. If this is suddenly changed, there will be audible clicks in the output.

Note that many opcodes (such as *delay* and *reverb*) are prepared for optional initialization. To use this feature, the *tival opcode* is suitable. Therefore, they need not be hidden by a *tigoto* jump.

Beginning with Csound version 3.53, strings are recognized in p-fields for opcodes that accept them (*convolve*, *adsyn*, *diskin*, etc.). There may be only one string per score line.

You can also turnoff notes from the score by using a negative number for the instrument (p1). This is equivalent to using the *turnoff2* opcode. When a note is turned off from the score, it is allowed to release (if *xtratim* or opcodes with release section like *linenr* are used) and only notes with the same fractional part are turned off. Also, only the last instance of the instrument will be turned off, so there have to be as many negative instrument numbers as positive ones for all notes to be turned off.

```
i 1.1  1  300  8.00
i 1.2  1  300  8.04
i 1.3  1  -300  8.07
i 1.3  1  -300  8.09
```

```

; notice that p-fields after p2 will be ignored if
; instrument number is negative
i -1.1 3 1 4.00
i -1.2 4 51 4.04
i -1.3 5 1 4.07
i -1.3 6 10 4.09

```

## Special Considerations

The maximum instrument number used to be 200. This has been changed to be limited by memory only (currently there is an internal soft limit of 200; this is automatically extended as required).

## Examples

Here is an example of the `i` statement. It uses the file `i_statement.csd` [examples/i\_statement.csd].

### Example 1167. Example of the `i` statement.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o i_statement.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 10

  icps      init      cpspch(p4)           ; Get target pitch from score event
  iportime  init      abs(p3)/7           ; Portamento time dep on note length
  iamp0     init      p5                   ; Set default amps
  iamp1     init      p5
  iamp2     init      p5

  itie      tival     ; Check if this note is tied,
  if itie == 1      igoto nofadein       ; if not fade in
  iamp0     init      0

nofadein:
  if p3 < 0      igoto nofadeout       ; Check if this note is held, if not fade out
  iamp2     init      0

nofadeout:
; Now do amp from the set values:
kamp      linseg   iamp0, .03, iamp1, abs(p3)-.03, iamp2

; Skip rest of initialization on tied note:
          tigoto   tieskip

  kcps      init      icps               ; Init pitch for untied note
  kcps      port     icps, iportime, icps ; Drift towards target pitch

  kpw       oscil    .4, rnd(1), 1, rnd(.7) ; A simple triangle-saw oscil
  ar        vco     kamp, kcps, 3, kpw+.5, 1, 1/icps

```

```
; (Used in testing - one may set ipch to cpspch(p4+2)
; and view output spectrum)
; ar oscil kamp, kcps, 1

outs ar, ar

tieskip: ; Skip some initialization on tied note

endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1 ; Sine

i10.1 0 -1 7.00 .15
i10.2 0 -1 7.04
i10.3 0 -1 7.07
i10.1 1 -1 8.00
i10.2 1 -1 8.04
i10.3 1 -1 8.07
i10.1 2 1 7.11
i10.2 2 1 8.04
i10.3 2 1 8.07
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Additional text (Csound Version 4.07) explaining tied notes, edited by Rasmus Ekman from a note by David Kirsh, posted to the Csound mailing list. Example instrument by Rasmus Ekman.

Updated August 2002 thanks to a note from Rasmus Ekman. There is no longer a hard limit of 200 instruments.

# m Statement (Mark Statement)

`m` — Sets a named mark in the score.

## Description

Sets a named mark in the score, which can be used by an *n statement*.

## Syntax

```
m pl
```

## Initialization

*pl* -- Name of mark.

## Performance

This can be helpful in setting a up verse and chorus structure in the score. Names may contain letters and numerals.

For example, the following score:

```
m foo
i1 0 1
i1 1 1.5
i1 2.5 2
s
i1 0 10
s
n foo
e
```

Will be passed from the preprocessor to Csound as:

```
i1 0 1
i1 1 1.5
i1 2.5 2
s
i1 0 10
s
;; this is named section repeated
i1 0 1
i1 1 1.5
i1 2.5 2
s
;; end of named section
e
```

## Examples

Here is an example of the `m` statement. It uses the file *m.csd* [examples/m.csd].

**Example 1168. Example of the `m` statement.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o m.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

aenv expseg .01, p3*0.25, 1, p3*0.75, 0.01
asig poscil3 .8*aenv, p4, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine wave

m foo ;mark section
i 1 0 1 110
i 1 1.5 1 220
i 1 3 1 440
i 1 4.5 1 880
s ;second section
i 1 0 2 110
i 1 2 2 220
s
n foo ;repeat marked section
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
April, 1998

New in Csound version 3.48

# n Statement

`n` — Repeats a section.

## Description

Repeats a section from the referenced *m statement*.

## Syntax

```
n pl
```

## Initialization

*pl* -- Name of mark to repeat.

## Performance

This can be helpful in setting a up verse and chorus structure in the score. Names may contain letters and numerals.

For example, the following score:

```
m foo
i1 0 1
i1 1 1.5
i1 2.5 2
s
i1 0 10
s
n foo
e
```

Will be passed from the preprocessor to Csound as:

```
i1 0 1
i1 1 1.5
i1 2.5 2
s
i1 0 10
s
;; this is named section repeated
i1 0 1
i1 1 1.5
i1 2.5 2
s
;; end of named section
e
```

## Examples

See the example for the *m statement*.

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
April 1998

New in Csound version 3.48

# q Statement

q statement — This statement may be used to quiet an instrument.

## Description

This statement may be used to quiet an instrument.

## Syntax

```
q p1 p2 p3
```

## Performance

*p1* -- Instrument number to mute/unmute.

*p2* -- Action time in beats.

*p3* -- determines whether the instrument is muted/unmuted. The value of 0 means the instrument is muted, other values mean it is unmuted.

Note that this does not affect instruments that are already running at time *p2*. It blocks any attempt to start one afterwards.

## Examples

Here is an example of the q statement. It uses the file *q.csd* [examples/q.csd].

### Example 1169. Example of the q statement.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o q.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

aenv expseg .01, p3*0.25, 1, p3*0.75, 0.01
asig poscil3 .8*aenv, p4, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1;sine wave
```

```
s
q 1 6 0 ;mute at 6 seconds in this section
i 1 0 2 110
i 1 3 2 220
i 1 6 2 440
i 1 9 2 880

s
q 1 6 1 ;unmute again at 6 seconds in this section
i 1 0 2 110
i 1 3 2 220
i 1 6 2 440
i 1 9 2 880
e
</CsScore>
</CsoundSynthesizer>
```

# r Statement (Repeat Statement)

r — Starts a repeated section.

## Description

Starts a repeated section, which lasts until the next *s*, *r* or *e statement*.

## Syntax

```
r p1 p2
```

## Initialization

*p1* -- Number of times to repeat the section.

*p2* -- Macro(name) to advance with each repetition (optional).

## Performance

In order that the sections may be more flexible than simple editing, the macro named *p2* is given the value of 1 for the first time through the section, 2 for the second, and 3 for the third. This can be used to change p-field parameters, or ignored.



### Warning

Because of serious problems of interaction with macro expansion, sections must start and end in the same file, and not in a macro.

## Examples

Here is an example of the *r* statement. It uses the file *r.csd* [examples/r.csd].

### Example 1170. Example of the *r* statement.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o r.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
```

```
instr 1
; The score's p4 parameter has the number of repeats.
kreps = p4
; The score's p5 parameter has our note's frequency.
kcps = p5

; Print the number of repeats.
printks "Repeated %i time(s).\n", 1, kreps

; Generate a nice beep.
a1 oscil 20000, kcps, 1
out a1
endin

</CsInstruments>
<CsScore>

; Table #1, a sine wave.
f 1 0 16384 10 1

; We'll repeat this section 6 times. Each time it
; is repeated, its macro REPS_MACRO is incremented.
r6 REPS_MACRO

; Play Instrument #1.
; p4 = the r statement's macro, REPS_MACRO.
; p5 = the frequency in cycles per second.
i 1 00.10 00.10 $REPS_MACRO 1760
i 1 00.30 00.10 $REPS_MACRO 880
i 1 00.50 00.10 $REPS_MACRO 440
i 1 00.70 00.10 $REPS_MACRO 220

; Marks the end of the section.
s

e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
April, 1998

New in Csound version 3.48

Example written by Kevin Conder

# s Statement

s — Marks the end of a section.

## Description

The *s statement* marks the end of a section.

## Syntax

s [time]

## Initialization

The first p-field *time* is optional and if present determines the end time (length in beats) of the section. This time must be after the end of the last event in the section otherwise it will have no effect. This can be used to create a pause before the beginning of the next section or to allow "always on" instruments such as effects to play by themselves for some length of time.

## Performance

Sorting of the *i statement*, *f statement* and *a statement* by action time is done section by section.

Time warping for the *t statement* is done section by section.

All action times within a section are relative to its beginning. A section statement establishes a new relative time of 0, but has no other reinitializing effects (e.g. stored function tables are preserved across section boundaries).

A section is considered complete when all action times and finite durations have been satisfied (i.e., the "length" of a section is determined by the last occurring action or turn-off). A section can be extended by the use of an *f0 statement* or by supplying the optional *p1* value to the *s statement*.

A section ending automatically invokes a purge of inactive instrument and data spaces.



### Note

- Since score statements are processed section by section, the amount of memory required depends on the maximum number of score statements in a section. Memory allocation is dynamic, and the user will be informed as extra memory blocks are requested during score processing.
- For the end of the final section of a score, the *s statement* is optional; the *e statement* may be used instead.

## Examples

Here is an example of the *s statement*. It uses the file *s.csd* [examples/s.csd].

**Example 1171. Example of the *s statement*.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o s.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

aenv expseg .01, p3*0.25, 1, p3*0.75, 0.01
asig poscil3 .8*aenv, p4, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1;sine wave

s ;first section
i 1 0 2 110
i 1 3 2 220
i 1 6 2 440
i 1 9 2 880

s ;second section
i 1 0 2 880
i 1 3 2 440
i 1 6 2 220
i 1 9 2 110
e
</CsScore>
</CsoundSynthesizer>
```

# t Statement (Tempo Statement)

t — Sets the tempo.

## Description

This statement sets the tempo and specifies the accelerations and decelerations for the current section. This is done by converting beats into seconds.

## Syntax

```
t p1 p2 p3 p4 ... (unlimited)
```

## Initialization

*p1* -- Must be zero.

*p2* -- Initial tempo on beats per minute.

*p3, p5, p7,...* -- Times in beats (in non-decreasing order).

*p4, p6, p8,...* -- Tempi for the referenced beat times.

## Performance

Time and Tempo-for-that-time are given as ordered couples that define points on a "tempo vs. time" graph. (The time-axis here is in beats so is not necessarily linear.) The beat-rate of a Section can be thought of as a movement from point to point on that graph: motion between two points of equal height signifies constant tempo, while motion between two points of unequal height will cause an *accelerando* or *ritardando* accordingly. The graph can contain discontinuities: two points given equal times but different tempi will cause an immediate tempo change.

Motion between different tempos over non-zero time is inverse linear. That is, an *accelerando* between two tempos M1 and M2 proceeds by linear interpolation of the single-beat durations from 60/M1 to 60/M2.

The first tempo given must be for beat 0.

A tempo, once assigned, will remain in effect from that time-point unless influenced by a succeeding tempo, i.e. the last specified tempo will be held to the end of the section.

A *t statement* applies only to the score section in which it appears. Only one *t statement* is meaningful in a section; it can be placed anywhere within that section. If a score section contains no *t statement*, then beats are interpreted as seconds (i.e. with an implicit *t 0 60* statement).

N.B. If the CSound command includes a *-t flag*, the interpreted tempo of all score *t statements* will be overridden by the command-line tempo.

## Examples

Here is an example of the t statement. It uses the file *t.csd* [examples/t.csd].

**Example 1172. Example of the t statement.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o t.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

aenv expseg .01, p3*0.25, 1, p3*0.75, 0.01
asig poscil3 .8*aenv, p4, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine wave

t 0 240 12 30 15 240 ;start tempo = 240

i 1 0 2 110 ;tempo = 240
i 1 3 2 220 ;slow down &
i 1 6 2 440 ;slow down &
i 1 9 2 880 ;slow down &
i 1 12 2 110 ;slow down to 30 at 12 seconds
i 1 15 2 220 ;speed up to 240 again
i 1 18 2 440 ;stay at tempo 240
i 1 21 2 880
e
</CsScore>
</CsoundSynthesizer>
```

# v Statement

v — Provides for locally variable time warping of score events.

## Description

The *v statement* provides for locally variable time warping of score events.

## Syntax

```
v p1
```

## Initialization

**p1** -- Time warp factor (must be positive).

## Performance

The *v statement* takes effect with the following *i statement*, and remains in effect until the next *v statement*, *s statement*, or *e statement*.

## Examples

The value of **p1** is used as a multiplier for the start times (**p2**) of subsequent *i statements*.

```
i1 0 1 ; note1
v2
i1 1 1 ; note2
```

In this example, the second note occurs two beats after the first note, and is twice as long.

Although the *v statement* is similar to the *t statement*, the *v statement* is local in operation. That is, *v* affects only the following notes, and its effect may be cancelled or changed by another *v statement*.

Carried values are unaffected by the *v statement* (see *Carry*).

```
i1 0 1
v2
i. + .
i. . .
```

In this example, the *v statement* has no effect.

## Examples

Here is an example of the *v statement*. It uses the file *v.csd* [examples/v.csd].

**Example 1173. Example of the v statement.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime output leave only the line below:
; -o v.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

aenv expseg .01, p3*0.25, 1, p3*0.75, 0.01
asig poscil3 .4*aenv, p4, 1
      outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine wave
;because note 3 and 5 are played simultaneously and are nearly of the same frequency,
;played together they will create a "beating" sound.

i 1 0 2 110 ; note1
v2
i 1 3 . 220 ; note2
i 1 6 . 110 ; note3
v1
i 1 9 . 880 ; note4
i 1 12 . 100 ; note5
e
</CsScore>
</CsoundSynthesizer>
```

In this example, note3 and note5 occur simultaneously, while note4 actually occurs before note3, that is, at its original place. Durations are unaffected.

# x Statement

x — Skip the rest of the current section.

## Description

This statement may be used to skip the rest of the current section.

## Syntax

**x** anything

## Initialization

All pfields are ignored.

## Examples

Here is an example of the x statement. It uses the file *x.csd* [examples/x.csd].

### Example 1174. Example of the x statement.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o x.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

aenv expseg .01, p3*0.25, 1, p3*0.75, 0.01
asig poscil3 .8*aenv, p4, 1
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine wave

s ;first section
i 1 0 2 110
i 1 3 2 220
i 1 6 2 440
i 1 9 2 880
s ;second section
x ;skip the rest
i 1 0 2 110 ;of this section

```

```
i 1 3 2 220
i 1 6 2 440
i 1 9 2 880
s ;but continue with this one
i 1 0 2 880
i 1 3 2 440
i 1 6 2 220
i 1 9 2 110
e
</CsScore>
</CsoundSynthesizer>
```

# y Statement (or Seed Statement)

a — Set seed for random numbers.

## Description

Set seed for random numbers, either from p1 or, if omitted, the clock.

## Syntax

y [p1]

## Initialization

p1 -- An integer value between 0 and  $2^{32}$  used as a new seed for random numbers. If omitted, the system clock value will be used instead.

## Performance

The tilde symbol ~ can be used in an expression wherever a number is permissible to use. Each ~ will evaluate to a random value between zero (0) and one (1). If there is no y statement in the score, the pseudo-random generator will yield the same numbers in each performance. If a fixed seed is given, a different predictable series of pseudo-random numbers will be generated from this seed. If a y statement is used without p1, the system clock will be used as a seed, yielding a different series of pseudo-random numbers for each performance.

## Examples

Here is an example of the y statement. It uses the file *y\_statement.csd* [examples/y\_statement.csd].

### Example 1175. Example of the y statement.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc     -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o 0dbfs.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
ksmps = 10
nchnls = 2
0dbfs = 1

instr 1
  print p4
endin
</CsInstruments>
<CsScore>
```

```
y  
  
i 1 0 1 [~ * 9 + 1]  
i 1 + 1 [~ * 9 + 1]  
i 1 + 1 [~ * 9 + 1]  
i 1 + 1 [~ * 9 + 1]  
e  
  
</CsScore>  
</CsoundSynthesizer>
```

Each time this example is run, different values between 1 and 10 will be displayed. The seed used will be displayed as well.

## Credits

Author: John ffitich, 2014

New in version 6.03

# { Statement

{ — Begins a non-sectional, nestable loop.

## Description

The *{* and *}* *statements* can be used to repeat a group of score statements. These loops do not constitute independent score sections and thus may repeat events within the same section. Multiple loops may overlap in time or be nested within each other.

## Syntax

```
{ p1 p2
```

## Initialization

*p1* -- Number of times to repeat the loop.

*p2* -- A macro name that is automatically defined at the beginning of the loop and whose value is advanced with each repetition (optional). The initial value is zero and the final value is (*p1* - 1).

## Performance

The *{ statement* is used in conjunction with the *}* *statement* to define repeating groups of other score events. A score loop begins with the *{ statement* which defines the number of repetitions and a unique macro name that will contain the current loop counter. The body of a loop can contain any number of other events (including sectional breaks) and is terminated by a *}* *statement* on its own line. The *}* *statement* takes no parameters.

The use of the term "loop" here does not imply any sort of temporal succession to the loop iterations. In other words, the *p2* values of the events inside of the loop are not automatically incremented by the length of the loop in each repetition. This is actually an advantage since it allows groups of simultaneous events to be easily defined as well. The loop macro can be used along with *score expressions* to increase the start times of events or to vary the events in any other way desired for each iteration. The macro is incremented by one for each repetition. Note that unlike the *r statement*, the value of the macro the first time through the loop is zero (0), not one (1). Therefore the final value is one less than the number of repetitions.

Score loops are a very powerful tool. While similar to the section repeat facility (the *r statement*), their chief advantage is that the score events in successive iterations of the loop are not separated by a section termination. Thus, it is possible to create multiple loops that overlap in time. Loops also can be nested within each other to a depth of 39 levels.



### Warning

Because of serious problems of interaction with macro expansion, loops must start and end in the same file, and not in a macro.

## Examples

Here are some examples of the *{* and *}* *statements*.

**Example 1176. Sequentially repeat a three-note phrase four times.**

```
{ 4 CNT
i1 [0.00 + 0.75 * $CNT.] 0.2 220
i1 [0.25 + 0.75 * $CNT.] . 440
i1 [0.50 + 0.75 * $CNT.] . 880
}
```

is interpreted as

```
i1 0.00 0.2 220
i1 0.25 . 440
i1 0.50 . 880

i1 0.75 0.2 220
i1 1.00 . 440
i1 1.25 . 880

i1 1.50 0.2 220
i1 1.75 . 440
i1 2.00 . 880

i1 2.25 0.2 220
i1 2.50 . 440
i1 2.75 . 880
```

**Example 1177. Create a group of simultaneous harmonic partials.**

In this example,  $p4$  is assumed to be the frequency of the note event.

```
{ 8 PARTIAL
i1 0 1 [100 * ($PARTIAL. + 1)]
}
```

is interpreted as

```
i1 0 1 100
i1 0 1 200
i1 0 1 300
i1 0 1 400
i1 0 1 500
i1 0 1 600
i1 0 1 700
i1 0 1 800
```

Here is a full example of the `{` and `}` statements. It uses the file `leftbrace.csd` [examples/leftbrace.csd].

**Example 1178. An example of nested loops to create several inharmonic sine clusters.**

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
```

```

-odac          -iadc      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o abs.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
nchnls = 2

gaReverbSend init 0

; a simple sine wave partial
instr 1
  idur  =      p3
  iamp  =      p4
  ifreq =      p5
  aenv  linseg 0.0, 0.1*idur, iamp, 0.6*idur, iamp, 0.3*idur, 0.0
  aosc  oscili aenv, ifreq, 1
        vincr  gaReverbSend, aosc
endin

; global reverb instrument
instr 2
  al, ar reverb gaReverbSend, gaReverbSend, 0.85, 12000
        outs   gaReverbSend+al, gaReverbSend+ar
        clear  gaReverbSend
endin

</CsInstruments>
<CsScore>
f1 0 4096 10 1

{ 4 CNT
  { 8 PARTIAL
    ; start time      duration      amplitude      frequency

    i1 [0.5 * $CNT.] [1 + ($CNT * 0.2)] [500 + (~ * 200)] [800 + (200 * $CNT.) + ($PARTIAL. * 20
  ]
  }
}

i2 0 6
e

</CsScore>
</CsoundSynthesizer>

```

## Credits

Author: Gabriel Maldonado

New in Csound version 3.52 (?). (Fixed in version 5.08).

## } Statement

} — Ends a non-sectional, nestable loop.

## Description

The *{* and *}* *statements* can be used to repeat a group of score statements. These loops do not constitute independent score sections and thus may repeat events within the same section. Multiple loops may overlap in time or be nested within each other.

## Syntax

```
}
```

## Initialization

All pfields are ignored.

## Performance

The *}* *statement* is used in conjunction with the *{* *statement* to define repeating groups of other score events. A score loop begins with the *{* *statement* which defines the number of repetitions and a unique macro name that will contain the current loop counter. The body of a loop can contain any number of other events (including sectional breaks) and is terminated by a *}* *statement* on its own line. The *}* *statement* takes no parameters.

See the documentation for the *{* *statement* for further details.

## Examples

See the examples in the entry for the *{* *statement*.

## Credits

Author: Gabriel Maldonado

New in Csound version 3.52 (?). (Fixed in version 5.08).

## GEN Routines

GEN routines are used as data generators for function tables. When a function table is created using the *f score statement* the GEN function is given as its fourth argument. A negative GEN number implies that the function is not rescaled, and maintains its original values.

## Sine/Cosine Generators:

- *GEN09* - Composite waveforms made up of weighted sums of simple sinusoids.
- *GEN10* - Composite waveforms made up of weighted sums of simple sinusoids.

- *GEN11* - Additive set of cosine partials.
- *GEN19* - Composite waveforms made up of weighted sums of simple sinusoids.
- *GEN30* - Generates harmonic partials by analyzing an existing table.
- *GEN33* - Generate composite waveforms by mixing simple sinusoids.
- *GEN34* - Generate composite waveforms by mixing simple sinusoids.

## Line/Exponential Segment Generators:

- *GEN05* - Constructs functions from segments of exponential curves.
- *GEN06* - Generates a function comprised of segments of cubic polynomials.
- *GEN07* - Constructs functions from segments of straight lines.
- *GEN08* - Generate a piecewise cubic spline curve.
- *GEN16* - Creates a table from a starting value to an ending value.
- *GEN25* - Construct functions from segments of exponential curves in breakpoint fashion.
- *GEN27* - Construct functions from segments of straight lines in breakpoint fashion.

## File Access GEN Routines:

- *GEN01* - Transfers data from a soundfile into a function table.
- *GEN23* - Reads numeric values from a text file.
- *GEN28* - Reads a text file which contains a time-tagged trajectory.
- *GEN43* - Loads a PVOCEX file containing a PV analysis.
- *GEN49* - Transfers data from an MP3 soundfile into a function table.

## Numeric Value Access GEN Routines

- *GEN02* - Transfers data from immediate pfields into a function table.
- *GEN17* - Creates a step function from given x-y pairs.
- *GEN52* - Creates an interleaved multichannel table from the specified source tables, in the format expected by the *ftconv* opcode.

## Window Function GEN Routines

- *GEN20* - Generates functions of different windows.

## Random Function GEN Routines

- *GEN21* - Generates tables of different random distributions.

- *GEN40* - Generates a random distribution using a distribution histogram.
- *GEN41* - Generates a random list of numerical pairs.
- *GEN42* - Generates a random distribution of discrete ranges of values.

## Waveshaping GEN Routines

- *GEN03* - Generates a stored function table by evaluating a polynomial.
- *GEN13* - Stores a polynomial whose coefficients derive from the Chebyshev polynomials of the first kind.
- *GEN14* - Stores a polynomial whose coefficients derive from Chebyshevs of the second kind.
- *GEN15* - Creates two tables of stored polynomial functions.

## Amplitude Scaling GEN Routines

- *GEN04* - Generates a normalizing function.
- *GEN12* - Generates the log of a modified Bessel function of the second kind.
- *GEN24* - Reads numeric values from another allocated function-table and rescales them.

## Mixing GEN Routines

- *GEN18* - Writes composite waveforms made up of pre-existing waveforms.
- *GEN31* - Mixes any waveform specified in an existing table.
- *GEN32* - Mixes any waveform, resampled with either FFT or linear interpolation.

## Pitch and Tuning GEN Routines

- *GEN51* - fills a table with a fully customized micro-tuning scale, in the manner of Csound opcodes *cpstun*, *cpstuni* and *cpstmid*.

## Named GEN Routines

Csound's GEN routines can be extended with GEN function plugins. There is currently a simple GEN plugin that provides exponential and hyperbolic tangent functions, and the *sone* function. There is also a generator called *farey* for the Farey sequence operations, and a Bézier curve generator. These GEN functions are not called by number, but by name.

- "*tanh*" - fills a table from a hyperbolic tangent formula.
- "*exp*" - fills a table from an exponential formula.
- "*sone*" - fills a table from a *sone* function formula.
- "*farey*" - fills a table from a Farey sequence.
- "*quadbezier*" - fills a table with a quadratic Bézier curve.

- "*wave*" - fills a table from a wavelet transform.
- "*padsynth*" - fills a table using the padsynth algorithm.

# GEN01

GEN01 — Transfers data from a soundfile into a function table.

## Description

This subroutine transfers data from a soundfile into a function table.

## Syntax

```
f# time size l filcod skiptime format channel
```

## Performance

*size* -- number of points in the table. Ordinarily a power of 2 or a power-of-2 plus 1 (see *f statement*); the maximum table size is 16777216 ( $2^{24}$ ) points. The allocation of table memory can be *deferred* by setting this parameter to 0; the size allocated is then the number of points in the file (probably not a power-of-2), and the table is not usable by normal oscillators, but it is usable by a *loscil* unit. The soundfile can also be mono or stereo.

*filcod* -- integer or character-string denoting the source soundfile name. An integer denotes the file *soundin.filcod*; a character-string (in double quotes, spaces permitted) gives the filename itself, optionally a full pathname. If not a full path, the file is sought first in the current directory, then in that given by the environment variable *SSDIR* (if defined) then by *SFDIR*. See also *soundin*.

*skiptime* -- begin reading at *skiptime* seconds into the file.

*channel* -- channel number to read in. 0 denotes read all channels.

*format* -- specifies the audio data-file format:

1 - 8-bit signed character	4 - 16-bit short integers
2 - 8-bit A-law bytes	5 - 32-bit long integers
3 - 8-bit U-law bytes	6 - 32-bit floats

If *format* = 0 the sample format is taken from the soundfile header, or by default from the CSound *-o* command-line flag.



### Note

- Reading stops at end-of-file or when the table is full. Table locations not filled will contain zeros.
- If *p4* is positive, the table will be post-normalized (rescaled to a maximum absolute value of 1 after generation). A negative *p4* will cause rescaling to be skipped.
- GEN01 also works with WAV and OGG and a dozen and more other sound formats; these file formats depend on *libsndfile*, see <http://www.mega-nerd.com/libsndfile/>

## Examples

Here is an example of the GEN01 routine. It uses the files *gen01.csd* [examples/gen01.csd] and several sound files.

### Example 1179. An example of the GEN01 routine.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gen01.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;plays deferred and non-deferred sounds with loscil

ifn = p4
ibas = 1

asig loscil 1, 1, ifn, ibas
outs asig, asig

endin

instr 2 ;plays only non-deferred sound

isnd = p4
aread line sr*p3, p3, 0 ;play this backward
asig tablei aread, isnd ;use table 1
outs asig, asig

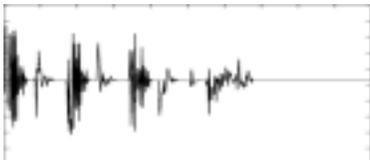
endin
</CsInstruments>
<CsScore>
f 1 0 131072 1 "beats.wav" 0 0 0 ;non-deferred sound
f 2 0 0 1 "flute.aiff" 0 0 0 ;& deferred sounds in
f 3 0 0 1 "beats.ogg" 0 0 0 ;different formats

i 1 0 1 1
i 1 + 1 2
i 1 + 1 3

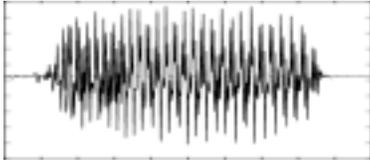
i 2 4 2 1 ;non-deferred sound for instr. 2
e
</CsScore>
</CsoundSynthesizer>

```

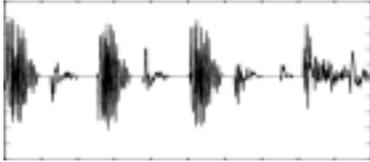
These are the diagrams of the waveforms of the GEN01 routines, as used in the example:



f 1 0 131072 1 "beats.wav" 0 0 0 - non-deferred sound



f 2 0 0 1 "flute.aiff" 0 0 0 - deferred sound



f 3 0 0 1 "beats.ogg" 0 0 0 - deferred sound

## Credits

September 2003. Thanks goes to Dr. Richard Boulanger for pointing out the references to the AIFF file format.

# GEN02

GEN02 — Transfers data from immediate pfields into a function table.

## Description

This subroutine transfers data from immediate pfields into a function table.

## Syntax

```
f # time size 2 v1 v2 v3 ...
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 or a power-of-2 plus 1 (see *f statement*). The maximum table size is 16777216 ( $2^{24}$ ) points.

*v1*, *v2*, *v3*, etc. -- values to be copied directly into the table space. The number of values is limited by the compile-time variable *PMAX*, which controls the maximum pfields (currently 1000). The values copied may include the table guard point; any table locations not filled will contain zeros.



### Note

If *p4* (the GEN routine number) is positive, the table will be post-normalized (rescaled to a maximum absolute value of 1 after generation). A negative *p4* will cause rescaling to be skipped. You will usually want to use -2 with this GEN function, so that your values are not normalized.

## Examples

Here is an example of the GEN02 routine. It uses the files *gen02.csd* [examples/gen02.csd].

### Example 1180. Example of the GEN02 routine.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o gen02.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
```

```

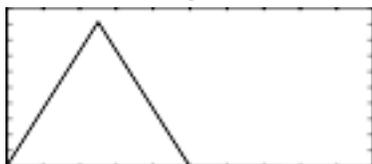
ifn = p4 ;choose different tables of GEN02
kcps init 1/p3 ;index over the length of entire note
kndx phasor kcps
ixmode = 1 ;normalize index data
kamp tablei kndx, ifn, ixmode
asig poscil kamp, 440, 1 ;use GEN02 as envelope for amplitude
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 8192 10 1 ;sine wave
f 2 0 5 2 0 2 0
f 3 0 5 2 0 2 10 0
f 4 0 9 2 0 2 10 100 0

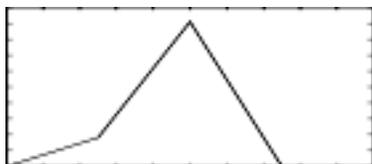
i 1 0 2 2
i 1 3 2 3
i 1 6 2 4
e
</CsScore>
</CsoundSynthesizer>

```

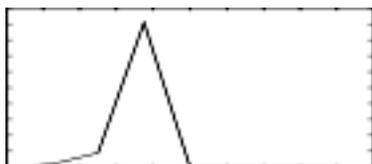
These are the diagrams of the waveforms of the GEN02 routines, as used in the example:



f 2 0 5 2 0 2 0



f 3 0 5 2 0 2 10 0



f 4 0 9 2 0 2 10 100 0

## See Also

*GEN17*

## Credits

December 2002. Thanks to Rasmus Ekman, corrected the limit of the *P*MAX variable.

# GEN03

GEN03 — Generates a stored function table by evaluating a polynomial.

## Description

This subroutine generates a stored function table by evaluating a polynomial in  $x$  over a fixed interval and with specified coefficients.

## Syntax

```
f # time size 3 xval1 xval2 c0 c1 c2 ... cn
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 or a power-of-2 plus 1.

*xval1*, *xval2* -- left and right values of the  $x$  interval over which the polynomial is defined ( $xval1 < xval2$ ). These will produce the 1st stored value and the (power-of-2 plus 1)th stored value respectively in the generated function table.

*c0*, *c1*, *c2*, ..., *cn* -- coefficients of the  $n$ th-order polynomial

$$C_0 + C_1x + C_2x^2 + \dots + C_nx^n$$

Coefficients may be positive or negative real numbers; a zero denotes a missing term in the polynomial. The coefficient list begins in p7, providing a current upper limit of 144 terms.



### Note

- The defined segment  $[fn(xval1), fn(xval2)]$  is evenly distributed. Thus a 512-point table over the interval  $[-1,1]$  will have its origin at location 257 (at the start of the 2nd half). Provided the extended guard point is requested, both  $fn(-1)$  and  $fn(1)$  will exist in the table.
- *GEN03* is useful in conjunction with *table* or *tablei* for audio waveshaping (sound modification by non-linear distortion). Coefficients to produce a particular formant from a sinusoidal lookup index of known amplitude can be determined at preprocessing time using algorithms such as Chebyshev formulae. See also *GEN13*.

## Examples

Here is an example of the GEN03 generator. It uses the file *gen03.csd* [examples/gen03.csd].

### Example 1181. Example of the GEN03 generator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac ;;realtime audio out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
```

```

; -o gen03.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
;example by Russell Pinkston - Univ. of Texas (but slightly modified)

gisine  ftgen 1, 0, 16384, 10, 1      ;sine wave

instr  1

ihertz = cpspch(p4)
ipkamp = p5
iwsfn = p6      ;waveshaping function
inmfn = p7      ;normalization function
aenv  linen 1, .01, p3, .1      ;overall amp envelope
actrl  linen 1, 2, p3, 2      ;waveshaping index control
aindex poscil actrl/2, ihertz, gisine      ;sine wave to be distorted
asignal tablei .5+aindex, iwsfn, 1      ;waveshaping
anormal tablei actrl, inmfn,1      ;amplitude normalization
asig = asignal*anormal*ipkamp*aenv
asig  dcblock2 asig      ;get rid of possible DC
outs  asig, asig

endin
</CsInstruments>
<CsScore>
; first four notes are specific Chebyshev polynomials using gen03. The values were obtained from Dodge

f4 0 513 3 -1 1 0 1      ; First-order Chebyshev: x
f5 0 257 4 4 1      ; Normalizing function for fn4

f6 0 513 3 -1 1 -1 0 2      ; Second-order Chebyshev: 2x2 - 1
f7 0 257 4 6 1      ; Normalizing function for fn6

f8 0 513 3 -1 1 0 -3 0 4      ; Third-order Chebyshev: 4x3 - 3x
f9 0 257 4 8 1      ; Normalizing function for fn8

f10 0 513 3 -1 10 0 -7 0 56 0 -112 0 64 ; Seventh-order Chebyshev: 64x7 - 112x5 + 56x3 - 7x
f11 0 257 4 10 1      ; Normalizing function for fn10

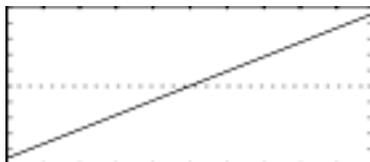
f12 0 513 3 -1 1 5 4 3 2 2 1      ; a 4th order polynomial function over the x-interval -1 to 1
f13 0 257 4 12 1      ; Normalizing function for fn12

; five notes with same fundamental, different waveshape & normalizing functions
;      pch  amp  wsfm  nmfn
i1 0 3 8.00      .7      4 5
i1 + . . . 6 7
i1 + . . . 8 9
i1 + . . . 10 11
i1 + . . . 12 13

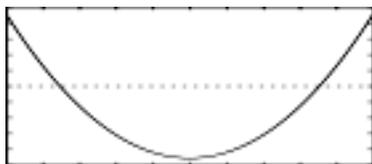
e
</CsScore>
</CsoundSynthesizer>

```

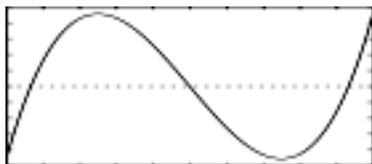
These are the diagrams of the waveforms of the GEN03 routines, as used in the example:



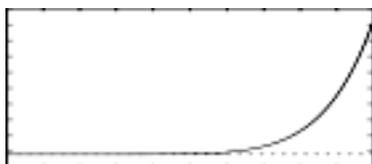
f4 0 513 3 1 1 0 1 - first-order Chebyshev:  $x$



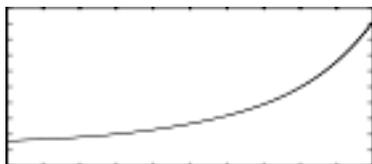
f6 0 513 3 -1 1 -1 0 2 - second-order Chebyshev:  $2x^2 - 1$



f8 0 513 3 -1 1 0 -3 0 4 - third-order Chebyshev:  $4x^3 - 3x$



f10 0 513 3 -1 10 0 -7 0 56 0 -112 0 64 - seventh-order Chebyshev:  $64x^7 - 112x^5 + 56x^3 - 7x$



f12 0 513 3 -1 1 5 4 3 2 2 1 - a 4th order polynomial function over the x-interval -1 to 1

## See Also

*GEN13*, *GEN14*, and *GEN15*.

Information about the Chebyshev polynomials on Wikipedia: [http://en.wikipedia.org/wiki/Chebyshev\\_polynomials](http://en.wikipedia.org/wiki/Chebyshev_polynomials)

# GEN04

GEN04 — Generates a normalizing function.

## Description

This subroutine generates a normalizing function by examining the contents of an existing table.

## Syntax

```
f # time size 4 source# sourcemode
```

## Initialization

*size* -- number of points in the table. Should be power-of-2 plus 1. Must not exceed (except by 1) the size of the source table being examined; limited to just half that size if the *sourcemode* is of type offset (see below).

*source #* -- table number of stored function to be examined.

*sourcemode* -- a coded value, specifying how the source table is to be scanned to obtain the normalizing function. Zero indicates that the source is to be scanned from left to right. Non-zero indicates that the source has a bipolar structure; scanning will begin at the mid-point and progress outwards, looking at pairs of points equidistant from the center.



### Note

- The normalizing function derives from the progressive absolute maxima of the source table being scanned. The new table is created left-to-right, with stored values equal to  $1/(\text{absolute maximum so far scanned})$ . Stored values will thus begin with  $1/(\text{first value scanned})$ , then get progressively smaller as new maxima are encountered. For a source table which is normalized (values  $\leq 1$ ), the derived values will range from  $1/(\text{first value scanned})$  down to 1. If the first value scanned is zero, that inverse will be set to 1.
- The normalizing function from *GEN04* is not itself normalized.
- *GEN04* is useful for scaling a table-derived signal so that it has a consistent peak amplitude. A particular application occurs in waveshaping when the carrier (or indexing) signal is less than full amplitude.

## Examples

Here is a simple example of the GEN04 routine. It uses the files *gen04.csd* [examples/gen04.csd].

### Example 1182. An example of the GEN04 routine.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
```

```

; For Non-realtime ouput leave only the line below:
; -o gen04.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gisine  ftgen 0, 0, 16384, 10, 1      ;sine wave

instr  1

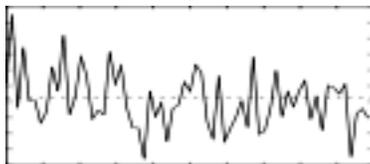
ihertz = cpspch(p4)
ipkamp = p5
iwsfn = p6      ;waveshaping function
inmfns = p7     ;normalization function
agate  linen 1, .01, p3, .1          ;overall amp envelope
kctrl  linen .9, 2, p3, 2          ;waveshaping index control
aindex poscil kctrl/2, ihertz, gisine ;sine wave to be distorted
asignal tablei .5+aindex, iwsfn, 1  ;waveshaping
knormal tablei 1/kctrl, inmfns, 1   ;amplitude normalization
outs   asignal*knormal*ipkamp*agate, asignal*knormal*ipkamp*agate

endin
</CsInstruments>
<CsScore>
f1 0 64 21 6 ;Gaussian (random) distribution
f2 0 33 4 1 1 ;normalizing function with midpoint bipolar offset

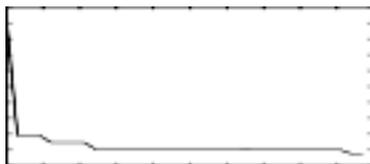
s
; st dur pch amp  wsfns inmfns
i1  0 4 6.00 .7 1 2
i1  4 . 7.00 .
i1  8 . 8.00 .
;-----
f3 0 1025 13 1 1 0 5 0 5 0 10 ;Chebyshev algorithm
f4 0 513 4 3 1 ;normalizing function with midpoint bipolar offset
s
; st dur pch amp  wsfns inmfns
i1  0 4 6.00 .9 3 4
i1  4 . 7.00 .
i1  8 . 8.00 .
e
</CsScore>
</CsSoundSynthesizer>

```

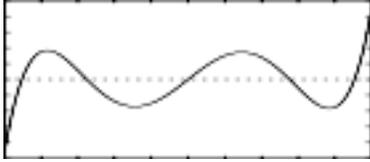
These are the diagrams of the waveforms of the GEN04 routines, as used in the example:



f1 0 64 21 6 - Gaussian (random) distribution



f2 0 33 4 1 1 - AND its normalizing function with midpoint bipolar offset



f3 0 1025 13 1 1 0 5 0 5 0 10 - Chebyshev algorithm



f4 0 513 4 3 1 - AND its normalizing function with midpoint bipolar offset

# GEN05

GEN05 — Constructs functions from segments of exponential curves.

## Description

Constructs functions from segments of exponential curves.

## Syntax

```
f # time size 5 a n1 b n2 c ...
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see *f statement*).

*a*, *b*, *c*, etc. -- ordinate values, in odd-numbered pfields p5, p7, p9, . . . These must be nonzero and must be alike in sign.

*n1*, *n2*, etc. -- length of segment (no. of storage locations), in even-numbered pfields. Cannot be negative, but a zero is meaningful for specifying discontinuous waveforms. The sum  $n1 + n2 + \dots$  will normally equal *size* for fully specified functions. If the sum is smaller, the function locations not included will be set to zero; if the sum is greater, only the first *size* locations will be stored. Note that the values are rounded to integers before use.



### Note

- If p4 is positive, functions are post-normalized (rescaled to a maximum absolute value of 1 after generation). A negative p4 will cause rescaling to be skipped.
- Discrete-point linear interpolation implies an increase or decrease along a segment by equal differences between adjacent locations; exponential interpolation implies that the progression is by equal ratio. In both forms the interpolation from *a* to *b* is such as to assume that the value *b* will be attained in the  $n + 1$ th location. For discontinuous functions, and for the segment encompassing the end location, this value will not actually be reached, although it may eventually appear as a result of final scaling.

## Examples

Here is a simple example of the GEN05 routine. It uses the files *gen05.csd* [examples/gen05.csd].

### Example 1183. An example of the GEN05 routine.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gen05.wav -W ;; for file output any platform
</CsOptions>
```

```

<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
Odbfs = 1

instr 1

ifn = p4      ;choose different tables for GEN05
kcps init 1/p3      ;index over the length of entire note
kndx phasor kcps
ixmode = 1      ;normalize index data
kamp tablei kndx, ifn, ixmode
asig poscil kamp, 440, 1
      outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 8192 10 1 ;sine wave
f 2 0 129 5 1 100 0.0001 29 ;short attack
f 3 0 129 5 0.00001 87 1 22 .5 20 0.0001 ;long attack

i 1 0 2 2
i 1 3 2 3

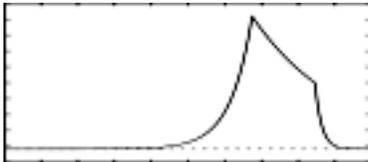
e
</CsScore>
</CsoundSynthesizer>

```

These are the diagrams of the waveforms of the GEN05 routines, as used in the example:



f 2 0 129 5 1 100 0.0001 29 - waveform that goes over 100 points from 1 to 0.0001, stay there for 29 points



f 3 0 129 5 0.00001 87 1 22 .5 20 0.0001 - waveform that goes from 0.00001 to 1 in 87 points, then from 1 to .5 in 22 points and then from .5 to 0.0001 in 20 points

## See Also

*GEN06*, *GEN07*, and *GEN08*

# GEN06

GEN06 — Generates a function comprised of segments of cubic polynomials.

## Description

This subroutine will generate a function comprised of segments of cubic polynomials, spanning specified points just three at a time.

## Syntax

```
f # time size 6 a n1 b n2 c n3 d ...
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see *f statement*).

*a, c, e, ...* -- local maxima or minima of successive segments, depending on the relation of these points to adjacent inflexions. May be either positive or negative.

*b, d, f, ...* -- ordinate values of points of inflexion at the ends of successive curved segments. May be positive or negative.

*n1, n2, n3 ...* -- number of stored values between specified points. Cannot be negative, but a zero is meaningful for specifying discontinuities. The sum  $n1 + n2 + \dots$  will normally equal *size* for fully specified functions. (for details, see *GEN05*).



### Note

*GEN06* constructs a stored function from segments of cubic polynomial functions. Segments link ordinate values in groups of 3: point of inflexion, maximum/minimum, point of inflexion. The first complete segment encompasses *b, c, d* and has length  $n2 + n3$ , the next encompasses *d, e, f* and has length  $n4 + n5$ , etc. The first segment (*a, b* with length *n1*) is partial with only one inflexion; the last segment may be partial too. Although the inflexion points *b, d, f ...* each figure in two segments (to the left and right), the slope of the two segments remains independent at that common point (i.e. the 1st derivative will likely be discontinuous). When *a, c, e ...* are alternately maximum and minimum, the inflexion joins will be relatively smooth; for successive maxima or successive minima the inflexions will be comb-like.

## Examples

Here is an example of the GEN06 routine. It uses the files *gen06.csd* [examples/gen06.csd].

### Example 1184. An example of the GEN06 routine.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc    ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
```

```

; -o gen06.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifn = p4 ;choose between tables
kcps init 1/p3 ;create index over duration of note.
kndx phasor kcps
ixmode = 1
kval table kndx, ifn, ixmode ;normalize mode

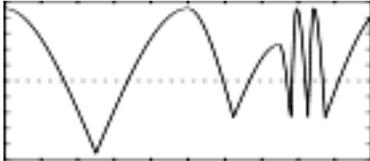
kfreq = kval * 30 ;scale frequency to emphasixe effect
asig poscil .7, 220 + kfreq, 1 ;add to frequency
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine wave.
f 2 0 513 6 1 128 -1 128 1 64 -.5 64 .5 16 -.5 8 1 16 -.5 8 1 16 -.5 84 1 16 -.5 8 .1 16 -.1 17 0
f 3 0 513 6 0 128 0.5 128 1 128 0 129 -1

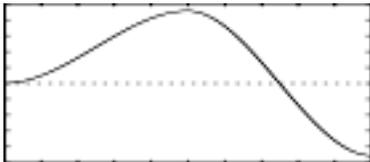
i 1 0 3 2
i 1 4 3 3
e
</CsScore>
</CsoundSynthesizer>

```

These are the diagrams of the waveforms of the GEN06 routines, as used in the example:



f 2 0 513 6 1 128 -1 128 1 64 -.5 64 .5 16 -.5 8 1 16 -.5 8 1 16 -.5 84 1 16 -.5 8 .1 16 -.1 17 0 - a not-so-smooth curve



f 3 0 513 6 0 128 0.5 128 1 128 0 129 -1 - a curve running 0 to 1 to -1, with a minimum, maximum and minimum at these values respectively. Inflexions are at .5 and 0 and are relatively smooth

## See Also

*GEN05*, *GEN07*, and *GEN08*

# GEN07

GEN07 — Constructs functions from segments of straight lines.

## Description

Constructs functions from segments of straight lines.

## Syntax

```
f # time size 7 a n1 b n2 c ...
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see *f statement*).

*a*, *b*, *c*, etc. -- ordinate values, in odd-numbered pfields p5, p7, p9, . . .

*n1*, *n2*, etc. -- length of segment (no. of storage locations), in even-numbered pfields. Cannot be negative, but a zero is meaningful for specifying discontinuous waveforms (e.g. in the example below). The sum *n1* + *n2* + . . . will normally equal *size* for fully specified functions. If the sum is smaller, the function locations not included will be set to zero; if the sum is greater, only the first *size* locations will be stored.



### Note

- If p4 is positive, functions are post-normalized (rescaled to a maximum absolute value of 1 after generation). A negative p4 will cause rescaling to be skipped.
- Discrete-point linear interpolation implies an increase or decrease along a segment by equal differences between adjacent locations; exponential interpolation implies that the progression is by equal ratio. In both forms the interpolation from *a* to *b* is such as to assume that the value *b* will be attained in the *n* + 1th location. For discontinuous functions, and for the segment encompassing the end location, this value will not actually be reached, although it may eventually appear as a result of final scaling.

## Examples

Here is an example of the GEN07 routine. It uses the files *gen07.csd* [examples/gen07.csd].

### Example 1185. An example of the GEN07 routine.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gen07.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
```

```

ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;use GEN07 to alter frequency

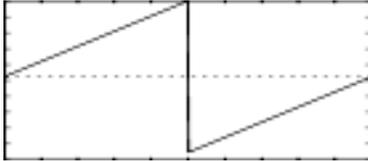
ifn = p4 ;use different GEN07 tables
kcps init 10/p3 ;index ftable 10 times over the duration of entire note
kndx phasor kcps
ixmode = 1 ;normalize index data
kfrq tablei kndx, ifn, ixmode
kfrq = kfrq*1000 ;scale
asig poscil .8, 1220+kfrq, 1 ;add to frequency
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 8192 10 1 ;sine wave
f 2 0 1024 7 0 512 1 0 -1 512 0 ;sawtooth up and down
f 3 0 1024 7 1 512 1 0 -1 512 -1 ;square
f 4 0 1024 7 1 1024 -1 ;saw down

i 1 0 2 2
i 1 + 2 3
i 1 + 1 4
e
</CsScore>
</CsoundSynthesizer>

```

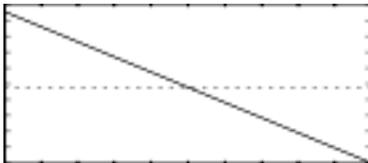
These are the diagrams of the waveforms of the GEN07 routines, as used in the example:



f 2 0 1024 7 0 512 1 0 -1 512 0 - sawtooth up and down, starting and ending at 0



f 3 0 1024 7 1 512 1 0 -1 512 -1 - a square from positive to negative



f 4 0 1024 7 1 1024 -1 - sawtooth down, a straight line from positive to negative

## See Also

*GEN05*, *GEN06*, and *GEN08*

# GEN08

GEN08 — Generate a piecewise cubic spline curve.

## Description

This subroutine will generate a piecewise cubic spline curve, the smoothest possible through all specified points.

## Syntax

```
f # time size 8 a n1 b n2 c n3 d ...
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see *f statement*).

*a, b, c, etc.* -- ordinate values of the function.

*n1, n2, n3 ...* -- length of each segment measured in stored values. May not be zero, but may be fractional. A particular segment may or may not actually store any values; stored values will be generated at integral points from the beginning of the function. The sum  $n1 + n2 + \dots$  will normally equal *size* for fully specified functions.



### Note

- *GEN08* constructs a stored table from segments of cubic polynomial functions. Each segment runs between two specified points but depends as well on their neighbors on each side. Neighboring segments will agree in both value and slope at their common point. (The common slope is that of a parabola through that point and its two neighbors). The slope at the two ends of the function is constrained to be zero (flat).
- *Hint*: to make a discontinuity in slope or value in the function as stored, arrange a series of points in the interval between two stored values; likewise for a non-zero boundary slope.

## Examples

Here is an example of the GEN08 routine. It uses the files *gen08.csd* [examples/gen08.csd].

### Example 1186. An example of the GEN08 routine.

```
<CsoundSynthesizer>
<CsoundOptions>
; Select audio/midi flags here according to platform
-odac ;;realtime audio out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o gen08.wav -W ;; for file output any platform
</CsoundOptions>
<CsoundInstruments>

sr = 44100
```

```

ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifn = p4 ;choose between tables
kcps init 1/p3 ;create index over duration of note.
kndx phasor kcps
ixmode = 1
kval table kndx, 2, ixmode ;normalize index data

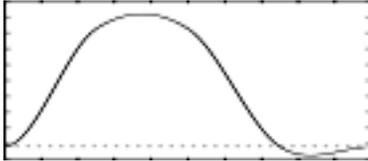
ibasefreq = 440
kfreq = kval * 100 ;scale
asig poscil .7, ibasefreq + kfreq, 1 ;and add to frequency
outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine wave.
f 2 0 65 8 0 16 1 16 1 16 0 17 0
f 3 0 65 8 -1 32 1 2 0 14 0 17 0

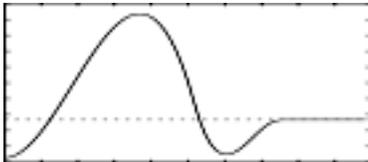
i 1 0 2 1
i 1 3 2 2
e
</CsScore>
</CsoundSynthesizer>

```

These are the diagrams of the waveforms of the GEN08 routines, as used in the example:



f 2 0 65 8 0 16 1 16 1 16 0 17 0 - a curve with a smooth hump in the middle, going briefly negative outside the hump then flat at its ends



f 3 0 65 8 -1 32 1 2 0 14 0 17 0 - from a negative value, a curve with a smooth hump, going negative creating a small hump then flat at its ends

## See Also

*GEN05*, *GEN06*, and *GEN07*

# GEN09

GEN09 — Generate composite waveforms made up of weighted sums of simple sinusoids.

## Description

These subroutines generate composite waveforms made up of weighted sums of simple sinusoids. The specification of each contributing partial requires 3 p-fields using *GEN09*.

## Syntax

```
f # time size 9 pna stra phsa pnb strb phsb ...
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see *f statement*).

*pna*, *pnb*, etc. -- partial no. (relative to a fundamental that would occupy *size* locations per cycle) of sinusoid a, sinusoid b, etc. Must be positive, but need not be a whole number, i.e., non-harmonic partials are permitted. Partial may be in any order.

*stra*, *strb*, etc. -- strength of partials *pna*, *pnb*, etc. These are relative strengths, since the composite waveform may be rescaled later. Negative values are permitted and imply a 180 degree phase shift.

*phsa*, *phsb*, etc. -- initial phase of partials *pna*, *pnb*, etc., expressed in degrees (0-360).



### Note

- These subroutines generate stored functions as sums of sinusoids of different frequencies. The two major restrictions on *GEN10* that the partials be harmonic and in phase do not apply to *GEN09* or *GEN19*.

In each case the composite wave, once drawn, is then rescaled to unity if *p4* was positive. A negative *p4* will cause rescaling to be skipped.

## Examples

Here is an example of the GEN09 routine. It uses the files *gen09.csd* [examples/gen09.csd].

### Example 1187. Example of the GEN09 routine.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
```

```

; -o gen09.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gi1 ftgen 1,0,2^10,9, 1,3,0, 3,1,0, 9,0.333,180 ;an approximation of a square wave
gi2 ftgen 2,0,2^10,9, 1,3,180, 3,1,0, 9,0.333,0 ;same values as gi1, except some phase values
gi3 ftgen 3,0,2^10,9, 1, .4, 0, 2.2, .5, 0, 3.8, 1, 0 ;inharmonic, but does not sound well --> wav
gi4 ftgen 4,0,2^10,9, 10, .4, 0, 22, .5, 0, 38, 1, 0 ;the same proportions, but value of partial n
;because the sudden "jump" like the one in gi3 will pop up only once in 10 repetitions

instr 1

kamp = .6
kcps = 220
ifn = p4

asig poscil kamp, kcps*p5, ifn
outs asig,asig

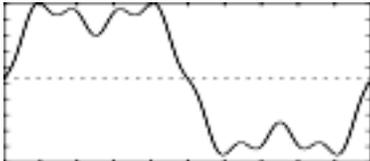
endin
</CsInstruments>
<CsScore>

i 1 0 2 1 1 ;subtle difference between table 1 and 2
i 1 3 2 2 1
i 1 7 2 3 1 ;big difference between table 3 and 4
i 1 10 2 4 .1 ;p5 has to compensate for the 10 repetitions of gi4 as opposed to gi3 to get the same pit

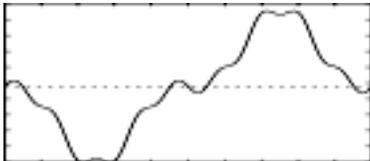
e
</CsScore>
</CsSoundSynthesizer>

```

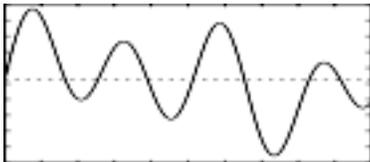
These are the diagrams of the waveforms of the GEN09 routines, as used in the example:



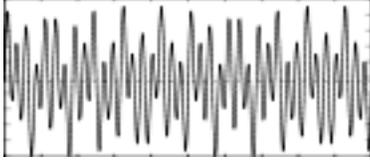
gi1 ftgen 1,0,2^10,9, 1,3,0, 3,1,0, 9,0.333,180 - approximation of a square wave



gi2 ftgen 2,0,2^10,9, 1,3,180, 3,1,0, 9,0.333,0 - same values as gi1, except phase value



gi3 ftgen 3,0,2^10,9, 1,2,0, 3,2,0, 9,0.333,180 - inharmonic partials, but with distortion due to the sudden jump in ending and beginning of the wave



gi4 ftgen 4,0,2^10,9, 1,2,180, 3,2,0, 9,0.333,0 - same ratio as gi3, except with less artefacts

## See Also

*GEN10, GEN19*

# GEN10

GEN10 — Generate composite waveforms made up of weighted sums of simple sinusoids.

## Description

These subroutines generate composite waveforms made up of weighted sums of simple sinusoids. The specification of each contributing partial requires 1 pfield using *GEN10*.

## Syntax

```
f # time size 10 str1 str2 str3 str4 ...
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see *f statement*).

*str1, str2, str3, etc.* -- relative strengths of the fixed harmonic partial numbers 1,2,3, etc., beginning in p5. Partial not required should be given a strength of zero.



### Note

- These subroutines generate stored functions as sums of sinusoids of different frequencies. The two major restrictions on *GEN10* that the partials be harmonic and in phase do not apply to *GEN09* or *GEN19*.

In each case the composite wave, once drawn, is then rescaled to unity if p4 was positive. A negative p4 will cause rescaling to be skipped.

## Examples

Here is an example of the GEN10 routine. It uses the files *gen10.csd* [examples/gen10.csd].

### Example 1188. An example of the GEN10 routine.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gen10.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kamp = .6
```

```

kcps = 440
ifn = p4

asig oscil kamp, kcps, ifn
outs asig,asig

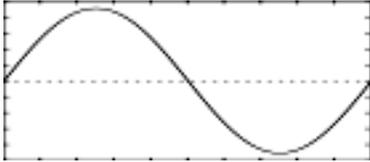
endin
</CsInstruments>
<CsScore>
f1 0 16384 10 1 ; Sine
f2 0 16384 10 1 0.5 0.3 0.25 0.2 0.167 0.14 0.125 .111 ; Sawtooth
f3 0 16384 10 1 0 0.3 0 0.2 0 0.14 0 .111 ; Square
f4 0 16384 10 1 1 1 0.7 0.5 0.3 0.1 ; Pulse

i 1 0 2 1
i 1 3 2 2
i 1 6 2 3
i 1 9 2 4

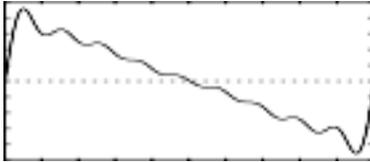
e
</CsScore>
</CsoundSynthesizer>

```

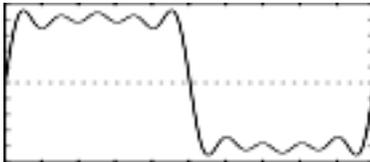
These are the diagrams of the waveforms of the GEN10 routines, as used in the example:



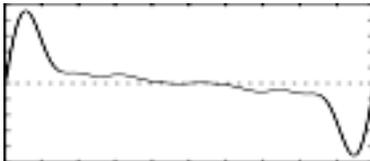
f 1 0 16384 10 1 - sine wave with only the fundamental frequency



f 2 0 16384 10 1 0.5 0.3 0.25 0.2 0.167 0.14 0.125 .111 - sawtooth, with a fundamental and 8 harmonics



f 3 0 16384 10 1 0 0.3 0 0.2 0 0.14 0 .111 - square wave, with a fundamental and 8 harmonics but 4 have 0 strength



f 4 0 16384 10 1 1 1 0.7 0.5 0.3 0.1 - pulse wave, with a fundamental and 8 harmonics

## See Also

*GEN09*, *GEN11*, and *GEN19*.

# GEN11

GEN11 — Generates an additive set of cosine partials.

## Description

This subroutine generates an additive set of cosine partials, in the manner of Csound generators *buzz* and *gbuzz*.

## Syntax

```
f # time size ll nh [lh] [r]
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see *f statement*).

*nh* -- number of harmonics requested. Must be positive.

*lh*(optional) -- lowest harmonic partial present. Can be positive, zero or negative. The set of partials can begin at any partial number and proceeds upwards; if *lh* is negative, all partials below zero will reflect in zero to produce positive partials without phase change (since cosine is an even function), and will add constructively to any positive partials in the set. The default value is 1

*r*(optional) -- multiplier in an amplitude coefficient series. This is a power series: if the *lh*th partial has a strength coefficient of *A* the (*lh* + *n*)th partial will have a coefficient of  $A * r^n$ , i.e. strength values trace an exponential curve. *r* may be positive, zero or negative, and is not restricted to integers. The default value is 1.



### Note

- This subroutine is a non-time-varying version of the CSound *buzz* and *gbuzz* generators, and is similarly useful as a complex sound source in subtractive synthesis. With *lh* and *r* present it parallels *gbuzz*; with both absent or equal to 1 it reduces to the simpler *buzz* (i.e. *nh* equal-strength harmonic partials beginning with the fundamental).
- Sampling the stored waveform with an oscillator is more efficient than using the dynamic buzz units. However, the spectral content is invariant and care is necessary, lest the higher partials exceed the Nyquist during sampling to produce fold-over.

## Examples

Here is an example of the GEN11 routine. It uses the files *gen11.csd* [examples/gen01.csd].

### Example 1189. An example of the GEN11 routine.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
```

```

; For Non-realtime ouput leave only the line below:
; -o gen11.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

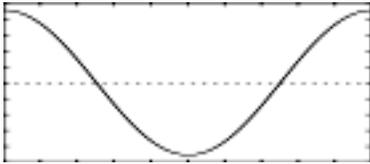
ifn = p4
asig oscil .8, 220, ifn
outs asig,asig

endin
</CsInstruments>
<CsScore>
f 1 0 16384 11 1 1 ;number of harmonics = 1
f 2 0 16384 11 10 1 .7 ;number of harmonics = 10
f 3 0 16384 11 10 5 2 ;number of harmonics = 10, 5th harmonic is amplified 2 times

i 1 0 2 1
i 1 + 2 2
i 1 + 2 3
e
</CsScore>
</CsoundSynthesizer>

```

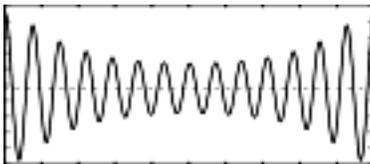
These are the diagrams of the waveforms of the GEN11 routines, as used in the example:



f 1 0 16384 11 1 1



f 2 0 16384 11 10 1 .7



f 3 0 16384 11 10 5 2

## See Also

*GEN10*

# GEN12

GEN12 — Generates the log of a modified Bessel function of the second kind.

## Description

This generates the log of a modified Bessel function of the second kind, order 0, suitable for use in amplitude-modulated FM.

## Syntax

```
f # time size 12 xint
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 or a power-of-2 plus 1 (see *f statement*). The normal value is power-of-2 plus 1.

*xint* -- specifies the *x* interval [0 to +*xint*] over which the function is defined.



### Note

- This subroutine draws the natural log of a modified Bessel function of the second kind, order 0 (commonly written as  $I$  subscript 0), over the *x*-interval requested. The call should have rescaling inhibited.
- The function is useful as an amplitude scaling factor in cycle-synchronous amplitude-modulated FM. (See Palamin & Palamin, *J. Audio Eng. Soc.*, 36/9, Sept. 1988, pp.671-684.) The algorithm is interesting because it permits the normally symmetric FM spectrum to be made asymmetric around a frequency other than the carrier, and is thereby useful for formant positioning. By using a table lookup index of  $I(r - 1/r)$ , where  $I$  is the FM modulation index and  $r$  is an exponential parameter affecting partial strengths, the Palamin algorithm becomes relatively efficient, requiring only *oscil*'s, table lookups, and a single *exp* call.

## Examples

Here is an example of the GEN12 generator. It uses the file *gen12.csd* [examples/gen12.csd].

### Example 1190. Example of the GEN12 generator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac ;;realtime audio out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gen12.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
;example from the Csound Book, page 87
sr = 44100
```

```

ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

idur   =      p3
iamp   =      p4
icarfrq =     p5
imodfrq =     p6
aenv   expseg .01, idur*.1, iamp, idur*.8, iamp*.75, idur*.1, .01
i1     =      p7*imodfrq ;p7=modulation index start
i2     =      p8*imodfrq ;p8=modulation index end
adev   line   i1, idur, i2 ;modulation frequency
aindex line   p7, idur, p8 ;modulation index

ar     linseg 1, .1, p9, p3-.2, p10, .1, 1 ; r value envelope: p9-p10 =exp. partial strength parameter
amp1   =      (aindex*(ar+(1/ar)))/2
afmod  oscili  amp1, imodfrq, 1 ;FM modulator (sine)
atab   =      (aindex*(ar-(1/ar)))/2 ;index to table
alook  tablei  atab, 37 ;table lookup to GEN12
aamod  oscili  atab, adev, 2 ;am modulator (cosine)
aamod  =      (exp(alook+aamod))*aenv
acar   oscili  aamod, afmod+icarfrq, 1 ;AFM (carrier)
asig   balance acar, aenv
outs   outs   asig, asig

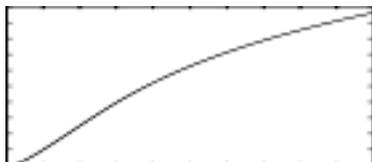
endin

</CsInstruments>
<CsScore>
f 1 0 8192 10 1
f 2 0 8192 9 1 1 90
f37 0 1024 -12 40 ;Bessel function-defined from 0 to 40

i 1 0 2 .2 800 800 1 6 .1 2
i 1 + . . 1900 147 8 1 4 .2
i 1 . . . 1100 380 2 9 .5 2
i 1 . 10 . 100 100 11 3 .2 5
s
i 1 0 1 .1 200 100 1 6 .1 2
i 1 + . < < < < < < <
i 1 + . . < < < < < < <
i 1 + . . < < < < < < <
i 1 + . . < < < < < < <
i 1 + . . < < < < < < <
i 1 + . . < < < < < < <
i 1 + 10 .2 800 800 9 1 .9 6
s
i 1 0 11 .25 50 51 1 6 .1 2
i 1 1 9 .05 700 401 1 6 .1 2
i 1 2 8 . 900 147 8 1 4 .2
i 1 3 7 . 1100 381 2 9 .5 2
i 1 4 6 . 200 102 11 3 .2 5
i 1 5 6 . 800 803 9 1 .9 6
e
</CsScore>
</CsoundSynthesizer>

```

This is the diagram of the waveform of the GEN12 routines, as used in the example:



f 37 0 1024 -12 40 - Bessel function-defined from 0 to 40

## Credits

Example is, with minor modifications, taken from The Csound Book (page 87).

# GEN13

GEN13 — Stores a polynomial whose coefficients derive from the Chebyshev polynomials of the first kind.

## Description

Uses Chebyshev coefficients to generate stored polynomial functions which, under waveshaping, can be used to split a sinusoid into harmonic partials having a pre-definable spectrum.

## Syntax

```
f # time size 13 xint xamp h0 h1 h2 ...
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 or a power-of-2 plus 1 (see *f statement*). The normal value is power-of-2 plus 1.

*xint* -- provides the left and right values  $[-xint, +xint]$  of the x interval over which the polynomial is to be drawn. These subroutines both call *GEN03* to draw their functions; the p5 value here is therefor expanded to a negative-positive p5, p6 pair before *GEN03* is actually called. The normal value is 1.

*xamp* -- amplitude scaling factor of the sinusoid input that is expected to produce the following spectrum.

*h0, h1, h2*, etc. -- relative strength of partials 0 (DC), 1 (fundamental), 2 ... that will result when a sinusoid of amplitude

$$xamp * \text{int}(\text{size}/2)/xint$$

is waveshaped using this function table. These values thus describe a frequency spectrum associated with a particular factor *xamp* of the input signal.

*GEN13* is the function generator normally employed in standard waveshaping. It stores a polynomial whose coefficients derive from the Chebyshev polynomials of the first kind, so that a driving sinusoid of strength *xamp* will exhibit the specified spectrum at output. Note that the evolution of this spectrum is generally not linear with varying *xamp*. However, it is bandlimited (the only partials to appear will be those specified at generation time); and the partials will tend to occur and to develop in ascending order (the lower partials dominating at low *xamp*, and the spectral richness increasing for higher values of *xamp*). A negative *hn* value implies a 180 degree phase shift of that partial; the requested full-amplitude spectrum will not be affected by this shift, although the evolution of several of its component partials may be. The pattern *+,+,-,-,+,+,...* for *h0,h1,h2...* will minimize the normalization problem for low *xamp* values (see above), but does not necessarily provide the smoothest pattern of evolution.

## Examples

Here is an example of the GEN13 generator. It uses the file *gen13.csd* [examples/gen13.csd].

**Example 1191. Example of the GEN13 generator.**

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gen13.wav -W   ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
;example by Russell Pinkston - Univ. of Texas (but slightly modified)

gisine   ftgen 0, 0, 16384, 10, 1      ;sine wave

instr 1

ihertz = cpspch(p4)
ipkamp = p5
iwsfn = p6      ;waveshaping function
inmfn = p7      ;normalization function
agate   linen 1, .01, p3, .1          ;overall amp envelope
kctrl   linen .99, 2, p3, 2          ;waveshaping index control
aindex  poscil kctrl/2, ihertz, gisine ;sine wave to be distorted
asignal tablei .5*aindex, iwsfn, 1    ;waveshaping
knormal tablei kctrl, inmfn, 1       ;amplitude normalization
asig    =      asignal*knormal*ipkamp*agate
outs   out    asig, asig

endin
</CsInstruments>
<CsScore>
; This proves the statement in Dodge (p. 147) that Chebyshev polynomials
; of order K have "only the kth harmonic." This is only true when the
; waveshaping index is at the maximum - i.e., when the entire transfer
; function is being accessed. RP.
;-----
; quasi sawtooth transfer function:
; h0 h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17
f1 0 513 13 1 1 0 100 -50 -33 25 20 -16.7 -14.2 12.5 11.1 -10 -9.09 8.333 7.69 -7.14 -6.67
f2 0 257 4 1 1 ; normalizing function with midpoint bipolar offset

; st dur pch amp wsfm nmfn
i1 0 4 6.00 .7 1 2
i1 4 . 7.00 .
i1 8 . 8.00 .
;-----
; quasi square wave transfer function:
; h0 h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17
f3 0 513 13 1 1 0 100 0 -33 0 20 0 -14.2 0 11.1 0 -9.09 0 7.69 0 -6.67
f4 0 257 4 3 1 ; normalizing function with midpoint bipolar offset

; st dur pch amp wsfm nmfn
i1 16 4 6.00 .7 3 4
i1 20 . 7.00 .
i1 24 . 8.00 .
;-----
; quasi triangle wave transfer function:

```

```

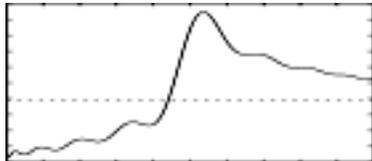
; h0 h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17
f5 0 513 13 1 1 0 100 0 -11.11 0 4 0 -2.04 0 1.23 0 -.826 0 .59 0 -.444
f6 0 257 4 5 1 ; normalizing function with midpoint bipolar offset
; st dur pch amp wsfm nmfn
i1 32 4 6.00 .7 5 6
i1 36 . 7.00 .
i1 40 . 8.00 .
;-----
; transfer function1: h0 h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
f7 0 513 13 1 1 0 1 -.8 0 .6 0 0 0 .4 0 0 0 0 .1 -.2 -.3 .5
f8 0 257 4 7 1 ; normalizing function with midpoint bipolar offset
; st dur pch amp wsfm nmfn
i1 48 4 5.00 .7 7 8
i1 52 . 6.00 .
i1 56 . 7.00 .
;-----
;=====
; This demonstrates the use of high partials, sometimes without a
; fundamental, to get quasi-inharmonic spectra from waveshaping.
;=====
; transfer function2: h0 h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
f9 0 513 13 1 1 0 0 0 -.1 0 .3 0 -.5 0 .7 0 -.9 0 1 0 -1 0
f10 0 257 4 9 1 ; normalizing function with midpoint bipolar offset
; st dur pch amp wsfm nmfn
i1 64 4 5.00 .7 9 10
i1 68 . 6.00 .
i1 72 . 7.00 .
;-----
; transfer function3: h0 h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14
f11 0 513 13 1 1 0 0 0 0 0 0 0 0 -1 0 1 0 0 0 -.1 0 .1
f12 0 257 4 11 1 ; normalizing function with midpoint bipolar offset

; st dur pch amp wsfm nmfn
i1 80 4 5.00 .7 11 12
i1 84 . 5.06 .
i1 88 . 6.00 .
;-----
;=====
; split a sinusoid into 3 odd-harmonic partials of relative strength 5:3:1
;=====
;-----
; transfer function4: h0 h1 h2 h3 h4 h5
f13 0 513 13 1 1 0 5 0 3 0 1
f14 0 257 4 13 1 ; normalizing function with midpoint bipolar offset

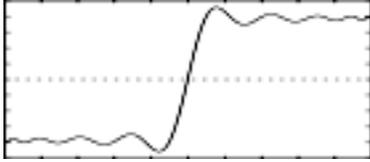
; st dur pch amp wsfm nmfn
i1 96 4 5.00 .7 13 14
i1 100 . 5.06 .
i1 104 . 6.00 .
e
</CsScore>
</CsoundSynthesizer>

```

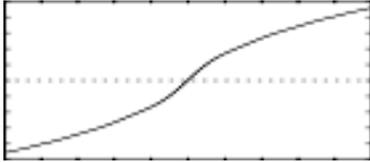
These are the diagrams of the waveforms of the GEN13 routines, as used in the example:



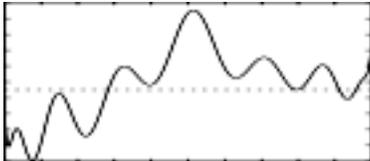
f1 0 513 13 1 1 0 100 -50 -33 25 20 -16.7 -14.2 12.5 11.1 -10 -9.09 8.333 7.69 -7.14 -6.67 6.25 5.88 -5.55  
-5.26 5 - quasi sawtooth transfer function



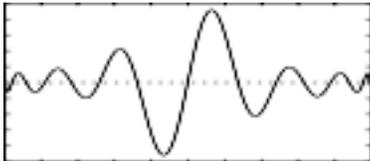
f3 0 513 13 1 1 0 100 0 -33 0 20 0 -14.2 0 11.1 0 -9.09 0 7.69 0 -6.67 0 5.88 0 -5.26 - quasi square wave transfer function



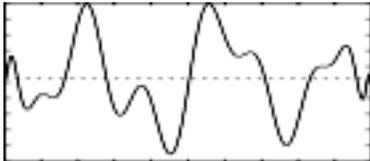
f5 0 513 13 1 1 0 100 0 -11.11 0 4 0 -2.04 0 1.23 0 -.826 0 .59 0 -.444 0 .346 0 -.277 - quasi triangle wave transfer function



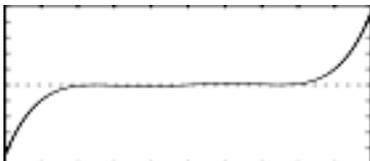
f7 0 513 13 1 1 0 1 -8.0 .6 0 0 0 .4 0 0 0 0 .1 -2 -.3 .5 - transfer function 1



f9 0 513 13 1 1 0 0 0 -1.0 .3 0 -.5 0 .7 0 -.9 0 1 0 -1 0 - transfer function 2



f11 0 513 13 1 1 0 0 0 0 0 0 -1 0 1 0 0 -.1 0 .1 0 -.2 .3 0 -.7 0 .2 0 -.1 - transfer function 3



f13 0 513 13 1 1 0 5 0 3 0 1 - split a sinusoid into 3 odd-harmonic partials of relative strength 5:3:1

## See Also

*GEN03*, *GEN14*, and *GEN15*.

Information about the Chebyshev polynomials on Wikipedia: [http://en.wikipedia.org/wiki/Chebyshev\\_polynomials](http://en.wikipedia.org/wiki/Chebyshev_polynomials)

# GEN14

GEN14 — Stores a polynomial whose coefficients derive from Chebyshevs of the second kind.

## Description

Uses Chebyshev coefficients to generate stored polynomial functions which, under waveshaping, can be used to split a sinusoid into harmonic partials having a pre-definable spectrum.

## Syntax

```
f # time size 14 xint xamp h0 h1 h2 ...
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 or a power-of-2 plus 1 (see *f statement*). The normal value is power-of-2 plus 1.

*xint* -- provides the left and right values  $[-xint, +xint]$  of the *x* interval over which the polynomial is to be drawn. These subroutines both call *GEN03* to draw their functions; the *p5* value here is therefore expanded to a negative-positive *p5, p6* pair before *GEN03* is actually called. The normal value is 1.

*xamp* -- amplitude scaling factor of the sinusoid input that is expected to produce the following spectrum.

*h0, h1, h2*, etc. -- relative strength of partials 0 (DC), 1 (fundamental), 2 ... that will result when a sinusoid of amplitude

$xamp * \text{int}(\text{size}/2)/xint$

is waveshaped using this function table. These values thus describe a frequency spectrum associated with a particular factor *xamp* of the input signal.



### Note

- *GEN13* is the function generator normally employed in standard waveshaping. It stores a polynomial whose coefficients derive from the Chebyshev polynomials of the first kind, so that a driving sinusoid of strength *xamp* will exhibit the specified spectrum at output. Note that the evolution of this spectrum is generally not linear with varying *xamp*. However, it is bandlimited (the only partials to appear will be those specified at generation time); and the partials will tend to occur and to develop in ascending order (the lower partials dominating at low *xamp*, and the spectral richness increasing for higher values of *xamp*). A negative *hn* value implies a 180 degree phase shift of that partial; the requested full-amplitude spectrum will not be affected by this shift, although the evolution of several of its component partials may be. The pattern  $+, +, -, -, +, +, \dots$  for *h0, h1, h2...* will minimize the normalization problem for low *xamp* values (see above), but does not necessarily provide the smoothest pattern of evolution.
- *GEN14* stores a polynomial whose coefficients derive from Chebyshevs of the second kind.

## Examples

Here is an example of the GEN14 generator. It uses the file *gen14.csd* [examples/gen14.csd].

### Example 1192. Example of the GEN14 generator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gen14.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
;after the example from The Csound Book, page 83

instr 1 ;compare results from GEN13 & GEN14

iwshpfn = p6
inrmfn = p7
aswp linseg 0.01, p3*.5, .49, p3*.5, 0.01 ;index sweep function
aindex poscil aswp, p5, 2 ;sound to waveshape
atable tablei aindex, iwshpfn, 1, .5 ;waveshape index
anrm tablei aswp*2, inrmfn, 1 ;normalization
aenv linen p4, .01, p3, .02 ;amplitude envelope
asig = (atable*anrm)*aenv ;normalize and impose envelope
asig dcblock2 asig ;get rid of DC
outs asig, asig

endin
</CsInstruments>
<CsScore>

f 2 0 8192 10 1 ;sine wave

f 28 0 4097 13 1 1 1 0 .8 0 .5 0 .2 ;waveshaping function: GEN13 - odd harmonics
f 280 0 2049 4 28 1 ;normalization function for f28
f 29 0 4097 14 1 1 1 0 .8 0 .5 0 .2 ;waveshaping function: GEN14 - same harmonics
f 290 0 2049 4 29 1 ;normalization function for f29

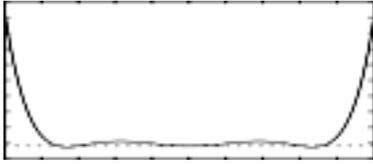
f 30 0 4097 13 1 1 0 1 0 .6 0 .4 0 .1 ;waveshaping function: GEN13 - even harmonics
f 301 0 2049 4 30 1 ;normalization function for f30
f 31 0 4097 14 1 1 0 1 0 .6 0 .4 0 .1 ;waveshaping function: GEN13 - even harmonics
f 310 0 2049 4 31 1 ;normalization function for 31

s
i1 0 3 .7 440 28 280
i1 4 . .7 . 29 290
i1 8 . .7 . 30 301
i1 12 3 .7 . 31 310

e
</CsScore>
</CsoundSynthesizer>

```

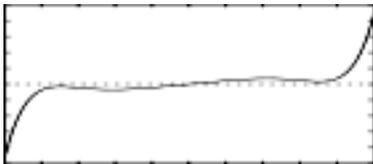
These are the diagrams of the waveforms of the GEN14 routines, as used in the example:



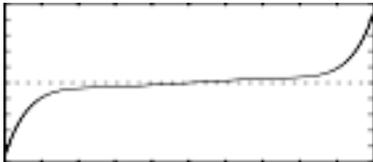
f28 0 4097 13 1 1 1 0 .8 0 .5 0 .2 - waveshaping function: GEN13, odd harmonics



f29 0 4097 14 1 1 1 1 0 .8 0 .5 0 .2 - waveshaping function: GEN14, the same odd harmonics



f30 0 4097 13 1 1 0 1 0 .6 0 .4 0 .1 - waveshaping function: GEN13, even harmonics



f31 0 4097 14 1 1 0 1 0 .6 0 .4 0 .1 - waveshaping function: GEN14, the same even harmonics

## See Also

*GEN03*, *GEN13*, and *GEN15*.

Information about the Chebyshev polynomials on Wikipedia: [http://en.wikipedia.org/wiki/Chebyshev\\_polynomials](http://en.wikipedia.org/wiki/Chebyshev_polynomials)

# GEN15

GEN15 — Creates two tables of stored polynomial functions.

## Description

This subroutine creates two tables of stored polynomial functions, suitable for use in phase quadrature operations.

## Syntax

```
f # time size 15 xint xamp h0 phs0 h1 phs1 h2 phs2 ...
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 or a power-of-2 plus 1 (see *f statement*). The normal value is power-of-2 plus 1.

*xint* -- provides the left and right values  $[-xint, +xint]$  of the  $x$  interval over which the polynomial is to be drawn. This subroutine will eventually call *GEN03* to draw both functions; this *p5* value is therefor expanded to a negative-positive *p5*, *p6* pair before *GEN03* is actually called. The normal value is 1.

*xamp* -- amplitude scaling factor of the sinusoid input that is expected to produce the following spectrum.

*h0*, *h1*, *h2*, ... *hn* -- relative strength of partials 0 (DC), 1 (fundamental), 2 ... that will result when a sinusoid of amplitude

$$xamp * \text{int}(\text{size}/2)/xint$$

is waveshaped using this function table. These values thus describe a frequency spectrum associated with a particular factor *xamp* of the input signal.

*phs0*, *phs1*, ... -- phase in degrees of desired harmonics *h0*, *h1*, ... when the two functions of *GEN15* are used with phase quadrature.



### Notes

*GEN15* creates two tables of equal size, labeled  $f \#$  and  $f \# + 1$ . Table  $\#$  will contain a Chebyshev function of the first kind, drawn using *GEN13* with partial strengths  $h0\cos(phs0)$ ,  $h1\cos(phs1)$ , ... Table  $\#+1$  will contain a Chebyshev function of the 2nd kind by calling *GEN14* with partials  $h1\sin(phs1)$ ,  $h2\sin(phs2)$ , ... (note the harmonic displacement). The two tables can be used in conjunction in a waveshaping network that exploits phase quadrature.

Before version 5.16 there was a bug (pointed out by Menno Knevel and fixed by François Pinot) on the number of *pfields* transmitted to *gen13* and *gen14* by *gen15*. The consequence is that all the *csd*, or *orc* and *sco* files that used *gen15* before this bug was fixed, are likely to sound different now.

## Examples

Here is an example of the *GEN15* routine. It uses the files *gen15.csd* [examples/gen15.csd].

**Example 1193. An example of the GEN15 routine.**

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gen15.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
;example from the Csound Book, page 85
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

idur = p3
iamp = p4
ifrq = cpspch(p5)      ;pitch
iswp1 = p6
iswp2 = p7
kswp line iswp1, p3, iswp2      ;amplitude sweep values
acosi oscili kswp*.5, ifrq, 2    ;f2=cosine wave
asine oscili kswp, ifrq, 1      ;f1=sine wave
atab1 tablei acosi, 33, 1, .5    ;tables a1 to GEN13
atab2 tablei acosi, 34, 1, .5    ;tables a1 to GEN14
knrm1 tablei kswp, 35, 1        ;normalizing f35
knrm2 tablei kswp, 36, 1        ;normalizing f36
anrm1 = atab1*knrm1             ;normalize GEN13 signal
anrm2 = atab2*knrm2*asine       ;normalize GEN14 signal
amix = anrm1+anrm2              ;mix GEN13 and GEN14
kenv expseg .001, idur*.1, iamp, idur*.1, iamp*.8, idur*.8, .001
asig = amix*kenv
outs asig, asig

endin

</CsInstruments>
<CsScore>
f 1 0 8193 10 1      ;sine wave
f 2 0 8193 9 1 1 90 ;cosine wave

; Note that all the f33 tables in the following sections are defined with p4=-15,
; which means that tables 33 and 34 will not be normalized. Thus if we display
; tables when running this example, we'll get correct diagrams even if one table
; has very small values instead of 0 values, due to cpu approximations in processing
; sin(180), as in sections 2, 4, and 5. This has no consequence on the audio result,
; because of the use of amp normalization (tables 35 and 36).

f 33 0 8193 -15 1 1 1 0 1 180 .8 45 .6 270 .5 90 .4 225 .2 135 .1 315 ;makes function tables 33 and 34
f 35 0 4097 4 33 1      ;amp normalization for f33
f 36 0 4097 4 34 1      ;amp normalization for f34
i 1 0 5 .6 8.00 0 1
i 1 + . .6 8.00 1 0
s
;even harmonics with no phase shift, odd harmonics with phase shift
f 33 0 8193 -15 1 1 1 0 1 0 1 180 1 180 1 0 1 0 1 180 1 180 1 0 1 0 1 180 1 180
f 35 0 4097 4 33 1      ;amp normalization for f33
f 36 0 4097 4 34 1      ;amp normalization for f34
i 1 0 5 .6 8.00 0 1
i 1 + . .6 8.00 1 0
s

```

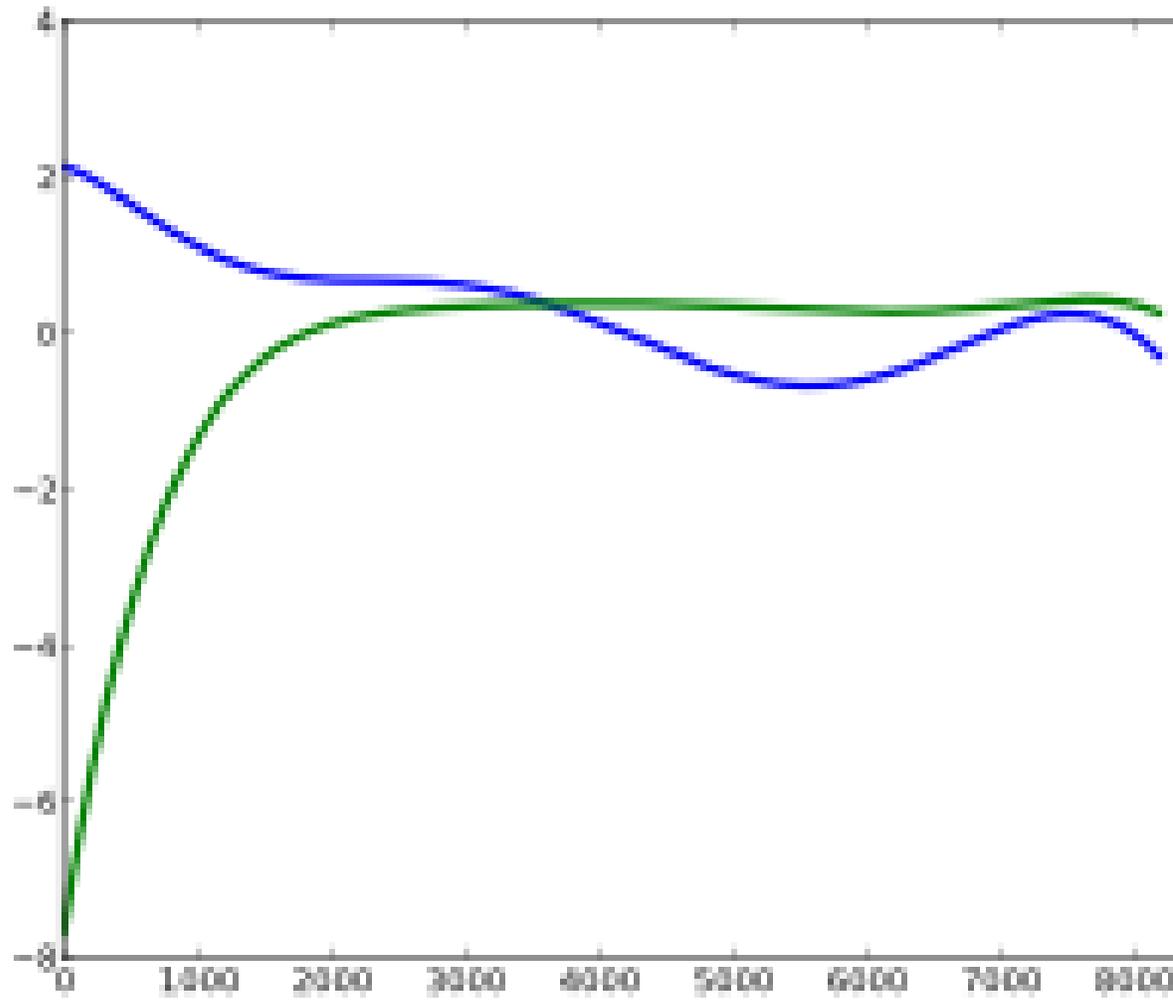
```

;different harmonic strenghts and phases
f 33 0 8193 -15 1 1 1 0 1 0 .9 180 .5 270 .75 90 .4 45 .2 225 .1 0
f 35 0 4097 4 33 1 ;amp normalization for f33
f 36 0 4097 4 34 1 ;amp normalization for f34
i 1 0 5 .6 8.00 0 1
i 1 + . .6 8.00 1 0
s
;lower harmonics no phase shift, upper harmonics with phase shift
f 33 0 8193 -15 1 1 1 0 1 0 .5 0 .9 0 .3 0 .75 0 .2 180 .6 180 .15 180 .5 180 .1 180
f 35 0 4097 4 33 1 ;amp normalization for f33
f 36 0 4097 4 34 1 ;amp normalization for f34
i 1 0 5 .6 8.00 0 1
i 1 + . .6 8.00 1 0

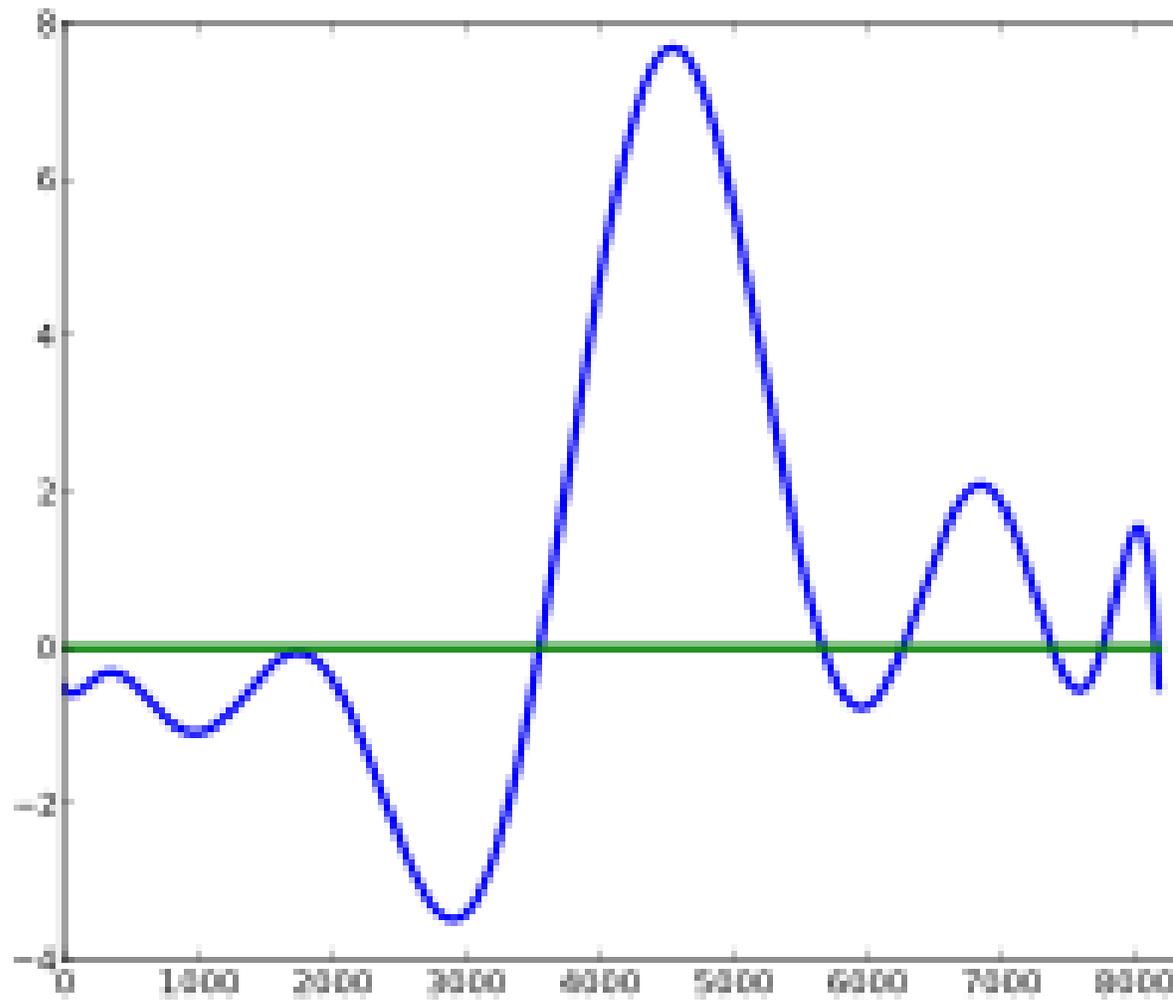
s
;lower harmonics with phase shift, upper harmonics no phase shift
f 33 0 8193 -15 1 1 1 180 1 180 .5 180 .9 180 .3 180 .75 180 .2 0 .6 0 .15 0 .5 0 .1 0
f 35 0 4097 4 33 1 ;amp normalization for f33
f 36 0 4097 4 34 1 ;amp normalization for f34
i 1 0 5 .6 8.00 0 1
i 1 + . .6 8.00 1 0
e
</CsScore>
</CsoundSynthesizer>

```

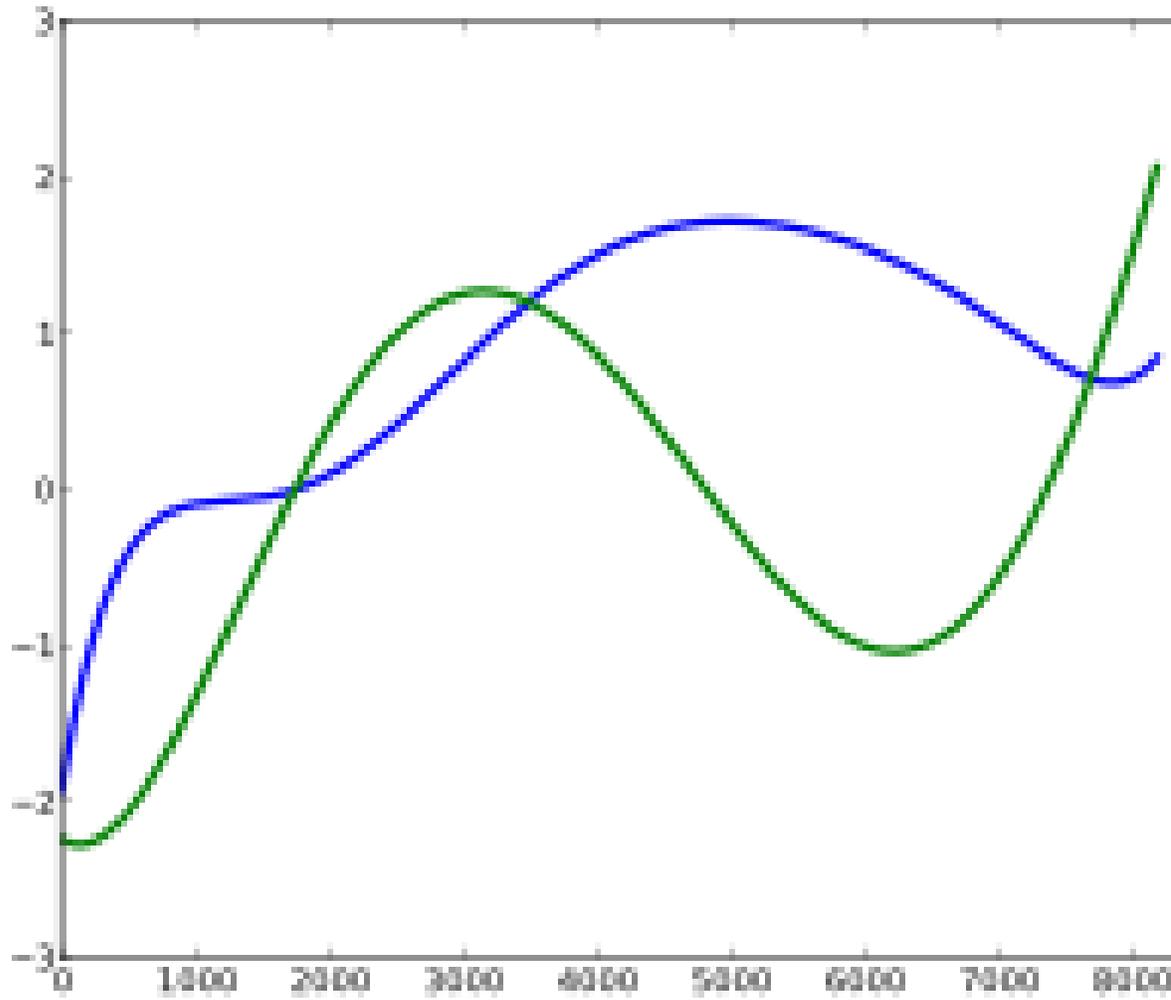
These are the diagrams of the waveforms of the GEN15 routine, as used in the example (in each diagram, the curve in blue is for ftable 33 and the curve in green is for ftable 34):



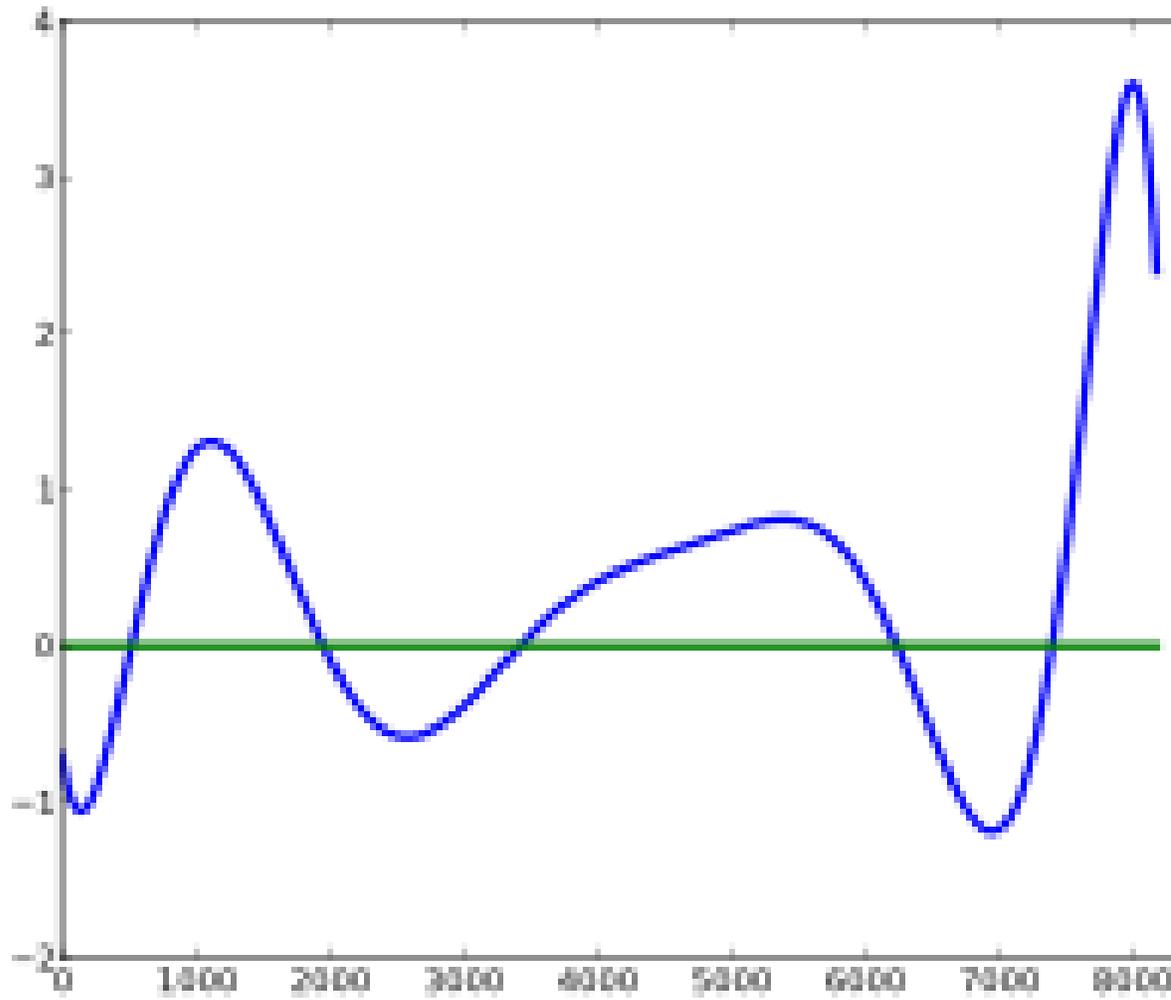
f 33 0 8193 -15 1 1 1 0 1 180 .8 45 .6 270 .5 90 .4 225 .2 135 .1 315



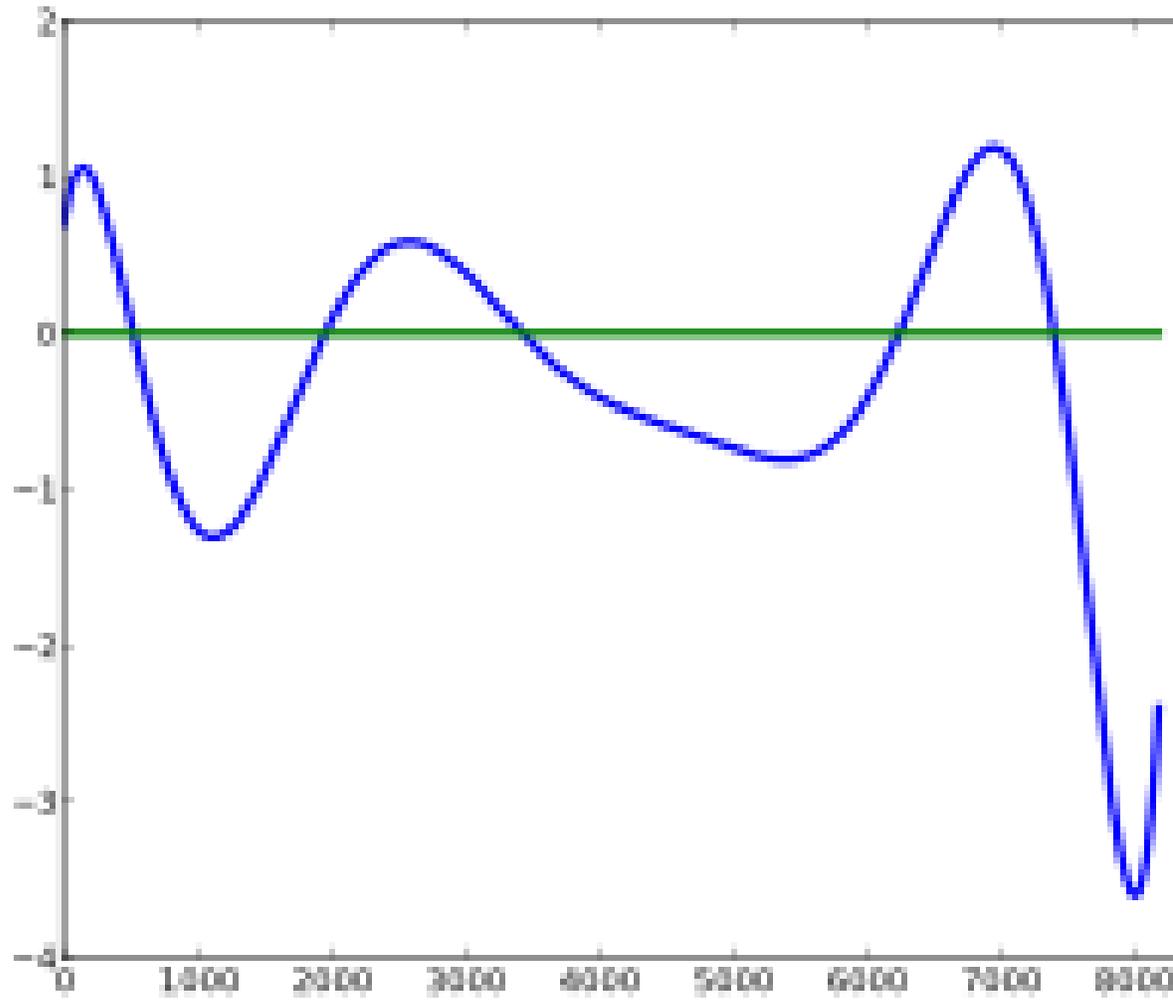
f 33 0 8193 -15 1 1 1 0 1 0 1 180 1 180 1 0 1 0 1 180 1 180 1 0 1 0 1 180 1 180



f 33 0 8193 -15 1 1 1 0 1 0 .9 180 .5 270 .75 90 .4 45 .2 225 .1 0



f 33 0 8193 -15 1 1 1 0 1 0.5 0.9 0.3 0.75 0.2 180 .6 180 .15 180 .5 180 .1 180



f 33 0 8193 -15 1 1 1 180 1 180 .5 180 .9 180 .3 180 .75 180 .2 0 .6 0 .15 0 .5 0 .1 0

## See Also

*GEN13*, and *GEN14*.

# GEN16

GEN16 — Creates a table from a starting value to an ending value.

## Description

Creates a table from *beg* value to *end* value of *dur* steps.

## Syntax

```
f # time size 16 val1 dur1 type1 val2 [dur2 type2 val3 ... typeX valN]
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 or a power-of-2 plus 1 (see *f statement*). The normal value is power-of-2 plus 1.



### Note

The end value is only reached when the table length is power-of-2 plus 1. (This length is crucial for use with the tab generator.)

*beg* -- starting value

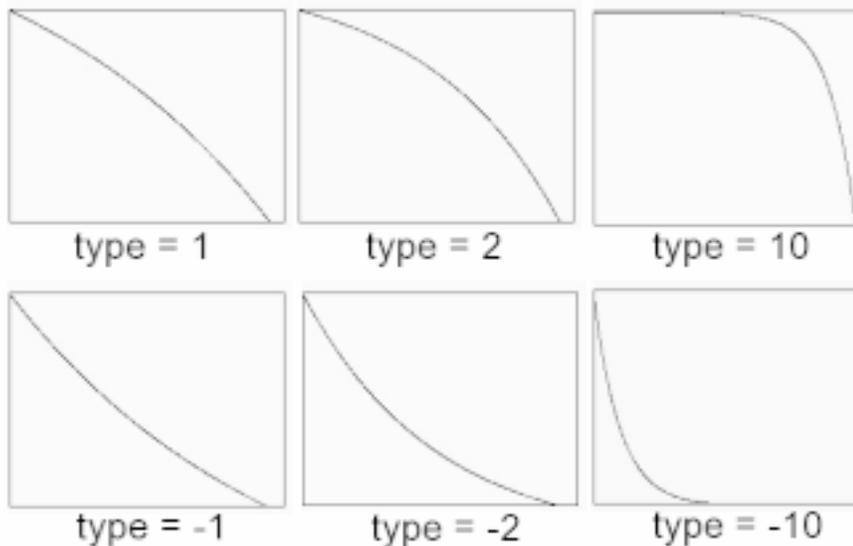
*dur* -- number of segments

*type* -- if 0, a straight line is produced. If non-zero, then *GEN16* creates the following curve, for *dur* steps:

$$\text{beg} + (\text{end} - \text{beg}) * (1 - \exp(i * \text{type} / (\text{dur} - 1))) / (1 - \exp(\text{type}))$$

*end* -- value after *dur* segments

Here are some examples of the curves generated for different values of *type*:



Tables generated by GEN16 for different values of type.



## Note

If  $type > 0$ , there is a slowly rising (concave) or slowly decaying (convex) curve, while if  $itype < 0$ , the curve is fast rising (convex) or fast decaying (concave). See also *transeg*.

### Example 1194. A simple example of the GEN16 routine.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in
-odac          -iadc    ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o gen16.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 128
nchnls = 1

instr 1
  kcps init 1/p3
  kndx phasor kcps

  ifn = p4
  ixmode = 1
  kval table kndx, ifn, ixmode

  ibasefreq = 440
  kfreq = kval * ibasefreq
  a1 oscil 20000, ibasefreq + kfreq, 1
  out a1
endin

</CsInstruments>
<CsScore>

f 1 0 16384 10 1

f 2 0 1024 16 1 1024 1 0
f 3 0 1024 16 1 1024 2 0
f 4 0 1024 16 1 1024 10 0
f 5 0 1024 16 1 1024 -1 0
f 6 0 1024 16 1 1024 -2 0
f 7 0 1024 16 1 1024 -10 0

i 1 0 2 2
i 1 + . 3
i 1 + . 4
i 1 + . 5
i 1 + . 6
i 1 + . 7

e

</CsScore>
</CsoundSynthesizer>

```

## Credits

Author: John ffitch  
University of Bath, Codemist. Ltd.  
Bath, UK  
October, 2000

New in Csound version 4.09

# GEN17

GEN17 — Creates a step function from given x-y pairs.

## Description

This subroutine creates a step function from given x-y pairs.

## Syntax

```
f # time size 17 x1 a x2 b x3 c ...
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 or a power-of-2 plus 1 (see *f statement*). The normal value is power-of-2 plus 1.

*x1*, *x2*, *x3*, etc. -- x-ordinate values, in ascending order, 0 first.

*a*, *b*, *c*, etc. -- y-values at those x-ordinates, held until the next x-ordinate.



### Note

This subroutine creates a step function of x-y pairs whose y-values are held to the right. The right-most y-value is then held to the end of the table. The function is useful for mapping one set of data values onto another, such as MIDI note numbers onto sampled sound ftable numbers ( see *loscil*).

## Examples

```
f 1 0 128 -17 0 1 12 2 24 3 36 4 48 5 60 6 72 7 84 8
```

This describes a step function with 8 successively increasing levels, each 12 locations wide except the last which extends its value to the end of the table. Rescaling is inhibited. Indexing into this table with a MIDI note-number would retrieve a different value every octave up to the eighth, above which the value returned would remain the same.

Here is a complete example of the GEN17 routine. It uses the files *gen17.csd* [examples/gen17.csd].

### Example 1195. An example of the GEN17 routine.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -M0 -+rtmidi=virtual ;;realtime audio out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gen17.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

inote cpsmidi
iveloc ampmidi .5
ictl midictrl 5 ;move slider of controller 5 to change ftable
itab table ictl, 2
aout poscil iveloc, inote, itab
outs aout, aout

endin
</CsInstruments>
<CsScore>
f 1 0 8193 10 1
f 2 0 128 -17 0 10 32 20 64 30 96 40 ;inhibit rescaling

f 10 0 16384 10 1 ; Sine
f 20 0 16384 10 1 0.5 0.3 0.25 0.2 0.167 0.14 0.125 .111; Sawtooth
f 30 0 16384 10 1 0 0.3 0 0.2 0 0.14 0 .111; Square
f 40 0 16384 10 1 1 1 1 0.7 0.5 0.3 0.1 ; Pulse

f 0 30 ;run for 30 seconds
e
</CsScore>
</CsoundSynthesizer>

```

This is the diagram of the waveform of the GEN17 routine, as used in the example:



f 2 0 128 -17 0 10 32 20 64 30 96 40 - a step function with 4 equal levels, each 32 locations wide except the last which extends its value to the end of the table

## See Also

*GEN02*

# GEN18

GEN18 — Writes composite waveforms made up of pre-existing waveforms.

## Description

Writes composite waveforms made up of pre-existing waveforms. Each contributing waveform requires 4 pfields and can overlap with other waveforms.

## Syntax

```
f # time size 18 fna ampa starta finisha fnb ampb startb finishb ...
```

## Initialization

*size* -- number of points in the table. Must be a power-of-2 (see *f* statement).

*fna, fnb, etc.* -- pre-existing table number to be written into the table.

*ampa, ampb, etc.* -- strength of wavefoms. These are relative strengths, since the composite waveform may be rescaled later. Negative values are permitted and imply a 180 degree phase shift.

*starta, startb, etc.* -- where to start writing the fn into the table.

*finisha, finishb, etc.* -- where to stop writing the fn into the table. The last available location is the power of two - 1.

## Examples

```
f 1 0 4096 10 1
f 2 0 1024 18 1 1 0 511 1 1 512 1023
```

f2 consists of two copies of f1 written in to locations 0-511 and 512-1023.

Here is an example of the GEN18 routine. It uses the files *gen18.csd* [examples/gen18.csd].

### Example 1196. An example of the GEN18 routine.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gen18.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
```

```

0dbfs = 1

instr 1

ifn = p4
ilen = ftlen(ifn)
aphase phasor 220
asig tablei aphase*ilen, ifn
outs asig, asig

endin
</CsInstruments>
<CsScore>

f 1 0 4096 10 1 ;sine
f 2 0 4096 10 1 0.5 0.3 0.25 0.2 0.167 0.14 0.125 .111 ;sawtooth
f 3 0 4096 11 10 5 2 ;cosine partials

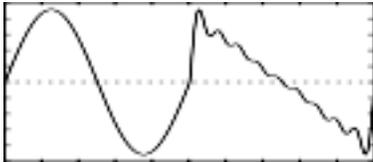
f 11 0 8192 18 1 1 0 4095 2 1 4096 8191 ;sine+sawtooth
f 12 0 8192 18 1 1 0 4095 3 1 4096 8191 ;sine+cosine partials
f 13 0 1024 18 1 0.7 0 767 3 0.7 512 1023 ;sine+cosine partials overlapped, shorter table

i 1 0 2 2 ;play sawtooth
i 1 + 2 3 ;then cosine partials
i 1 5 2 11 ;now sine+sawtooth
i 1 + 2 12 ;and sine+cosine partials
i 1 + 2 13 ;and sine+cosine partials overlapped, shorter table

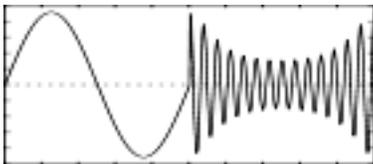
e
</CsScore>
</CsoundSynthesizer>

```

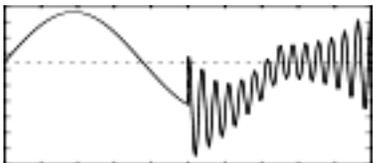
These are the diagrams of the waveforms generated by GEN18, as used in the example:



f 11 0 8193 18 1 1 0 4096 2 1 4097 8192 - composite waveform made up of a sine wave and a sawtooth



f 12 0 8192 18 1 1 0 4096 3 1 4097 8192 - composite waveform made up of a sine wave and a wave of cosine partials



f 13 0 1024 18 1 0.7 0 767 3 0.7 512 1023 - sine+cosine partials overlapped, also shorter table than f12

## Deprecated Names

*GEN18* was called *GEN22* in version 4.18. The name was changed due to a conflict with DirectCsound.

## Credits

Author: William "Pete" Moss  
University of Texas at Austin  
Austin, Texas USA  
January 2002

New in version 4.18, changed in version 4.19

# GEN19

GEN19 — Generate composite waveforms made up of weighted sums of simple sinusoids.

## Description

These subroutines generate composite waveforms made up of weighted sums of simple sinusoids. The specification of each contributing partial requires 4 p-fields using *GEN19*.

## Syntax

```
f # time size 19 pna  stra phsa dcoa pnb strb phsb dcob ...
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see *f statement*).

*pna*, *pnb*, etc. -- partial no. (relative to a fundamental that would occupy *size* locations per cycle) of sinusoid a, sinusoid b, etc. Must be positive, but need not be a whole number, i.e., non-harmonic partials are permitted. Partial numbers may be in any order.

*stra*, *strb*, etc. -- strength of partials *pna*, *pnb*, etc. These are relative strengths, since the composite waveform may be rescaled later. Negative values are permitted and imply a 180 degree phase shift.

*phsa*, *phsb*, etc. -- initial phase of partials *pna*, *pnb*, etc., expressed in degrees.

*dcoa*, *dcob*, etc. -- DC offset of partials *pna*, *pnb*, etc. This is applied *after* strength scaling, i.e. a value of 2 will lift a 2-strength sinusoid from range [-2,2] to range [0,4] (before later rescaling).



### Note

- These subroutines generate stored functions as sums of sinusoids of different frequencies. The two major restrictions on *GEN10* that the partials be harmonic and in phase do not apply to *GEN09* or *GEN19*.

In each case the composite wave, once drawn, is then rescaled to unity if *p4* was positive. A negative *p4* will cause rescaling to be skipped.

## Examples

Here is an example of the GEN19 routine. It uses the files *gen19.csd* [examples/gen19.csd].

### Example 1197. An example of the GEN19 routine.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o gen19.wav -W ;; for file output any platform
```

```

</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

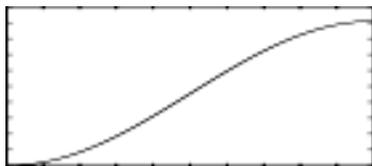
;p4 = transposition factor
;p5 = speed factor
;p6 = function table for grain data
i1 = sr/ftlen(1) ;scaling to reflect sample rate and table length
a1 phasor i1*p5 ;index for speed
asigl fog .5, 15, p4, a1, 1, 0, .01, .5, .01, 30, 1, 2, p3 ;left channel
asigr fog .4, 25, p4+.2, a1, 1, 0, .01, .5, .01, 30, 1, 2, p3, .5 ;right channel
outs asigl, asigr
endin

</CsInstruments>
<CsScore>
f 1 0 131072 1 "fox.wav" 0 0 0
f 2 0 1024 19 .5 .5 270 .5

i 1 0 10 .7 .1
i 1 + 4 1.2 2
e
</CsScore>
</CsoundSynthesizer>

```

This is the diagram of the waveform of the GEN19 routines, as used in the example:



f 2 0 1024 19 .5 .5 270 .5 - a rising sigmoid

## See Also

*GEN09* and *GEN10*

# GEN20

GEN20 — Generates functions of different windows.

## Description

This subroutine generates functions of different windows. These windows are usually used for spectrum analysis or for grain envelopes.

## Syntax

```
f # time size 20 window max [opt]
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 (+ 1).

*window* -- Type of window to generate:

- 1 = Hamming
- 2 = Hanning
- 3 = Bartlett ( triangle)
- 4 = Blackman ( 3-term)
- 5 = Blackman - Harris ( 4-term)
- 6 = Gaussian
- 7 = Kaiser
- 8 = Rectangle
- 9 = Sync

*max* -- For negative p4 this will be the absolute value at window peak point. If p4 is positive or p4 is negative and p6 is missing the table will be post-rescaled to a maximum value of 1.

*opt* -- Optional argument required by the Gaussian window and the Kaiser window.

## Examples

```
f      1      0      1024      20      5
```

This creates a function which contains a 4 - term Blackman - Harris window with maximum value of 1.

```
f      1      0      1024      -20      2      456
```

This creates a function that contains a Hanning window with a maximum value of 456.

```
f      1      0      1024      -20      1
```

This creates a function that contains a Hamming window with a maximum value of 1.

```
f      1      0      1024      20      7      1      2
```

This creates a function that contains a Kaiser window with a maximum value of 1. The extra argument specifies how "open" the window is, for example a value of 0 results in a rectangular window and a value of 10 in a Hamming like window.

```
f      1      0      1024      20      6      1      2
```

This creates a function that contains a Gaussian window with a maximum value of 1. The extra argument specifies how broad the window is, as the standard deviation of the curve; in this example the s.d. is 2. The default value is 1.

For all diagrams, see *Window Functions*

Here is an example of the GEN20 routine. It uses the file *gen20.csd* [examples/gen20.csd].

### Example 1198. Example of the GEN20 routine.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc      ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gen20.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

insnd = 10      ;"fox.wav"
ibasfrq = 44100 / ftlen(insnd) ;use original sample rate of insnd file

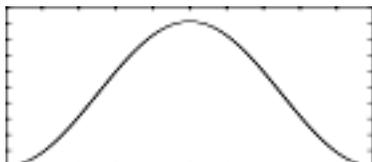
kamp expseg .001, p3/2, .7, p3/2, .8 ;envelope
kpitch line ibasfrq, p3, ibasfrq * .8
kdens line 600, p3, 10
kaoff line 0, p3, .1
kpoff line 0, p3, ibasfrq * .5
kgdur line .04, p3, .001 ;shorten duration of grain during note
imaxgdur = .5
igfn = p4      ;different windows
asigL grain kamp, kpitch, kdens, kaoff, kpoff, kgdur, insnd, igfn, imaxgdur, 0.0
asigR grain kamp, kpitch, kdens, kaoff, kpoff, kgdur, insnd, igfn, imaxgdur, 0.0
outs asigL, asigR

endin
</CsInstruments>
<CsScore>
```

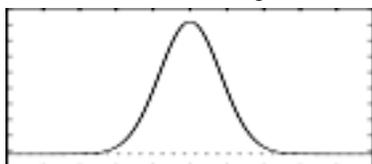
```
f1 0 512 20 2 ;Hanning window
f2 0 512 20 6 1 ;Gaussian window
f10 0 16384 1 "fox.wav" 0 0 0

i1 0 5 1 ;use Hanning window
i1 + 5 2 ;use Gaussian window
e
</CsScore>
</CsoundSynthesizer>
```

These are the diagrams of the waveforms of the GEN20 routines, as used in the example:



f 1 0 512 20 2 - Hanning window



f 2 0 512 20 6 1 - Gaussian window

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK

New in Csound version 3.2

Optional argument to Gaussian added in 5.10

# GEN21

GEN21 — Generates tables of different random distributions.

## Description

This generates tables of different random distributions. (See also *betarand*, *bexprnd*, *cauchy*, *exprand*, *gauss*, *linrand*, *pcauchy*, *poisson*, *trirand*, *unirand*, and *weibull*)

## Syntax

```
f # time size 21 type level [arg1 [arg2]]
```

## Initialization

*time* and *size* are the usual GEN function arguments. *level* defines the amplitude. Note that GEN21 is not self-normalizing as are most other GEN functions. *type* defines the distribution to be used as follow:

- 1 = Uniform (positive numbers only)
- 2 = Linear (positive numbers only)
- 3 = Triangular (positive and negative numbers)
- 4 = Exponential (positive numbers only)
- 5 = Biexponential (positive and negative numbers)
- 6 = Gaussian (positive and negative numbers)
- 7 = Cauchy (positive and negative numbers)
- 8 = Positive Cauchy (positive numbers only)
- 9 = Beta (positive numbers only)
- 10 = Weibull (positive numbers only)
- 11 = Poisson (positive numbers only)

Of all these cases only 9 (Beta) and 10 (Weibull) need extra arguments. Beta needs two arguments and Weibull one.

If *type* = 6, the random numbers in the ftable follow a normal distribution centered around 0.0 ( $\mu = 0.0$ ) with a variance (sigma) of  $level / 3.83$ . Thus more than 99.99% of the random values generated are in the range  $-level$  to  $+level$ . The default value for *level* is 1 (sigma = 0.261). If a mean value different of 0.0 is desired, this mean value has to be added to the generated numbers.

## Examples

```
f1 0 1024 21 1 ; Uniform (white noise)
f1 0 1024 21 6 ; Gaussian (mu=0.0, sigma=1/3.83=0.261)
f1 0 1024 21 6 5.745 ; Gaussian (mu=0.0, sigma=5.745/3.83=1.5)
```

```
f1 0 1024 21 9 1 1 2 ; Beta (note that level precedes arguments)
f1 0 1024 21 10 1 2 ; Weibull
```

All of the above additions were designed by the author between May and December 1994, under the supervision of Dr. Richard Boulanger.

Here is a complete example of the GEN21 routine. It uses the file *gen21.csd* [examples/gen21.csd].

### Example 1199. Example of the GEN21 routine.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac ;;realtime audio out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gen21.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifn = p4
isize = ftlen(ifn)
prints "TABLE NUMBER: %d\n", ifn
prints "Index\tValue\n"

iindex = 0 ;start loop
begin_loop:
  ivalue tab_i iindex, ifn
  prints "%d:\t%f\n", iindex, ivalue
  iindex = iindex + 1
  if (iindex < isize) igoto begin_loop

;produce sound - and repeat it 10 times so you can hear the patterns:
aphase phasor 10/10 ;play all 32 values 10x over 10 seconds
aphase = aphase*isize ;step through table
afrq table aphase, p4 ;read table number
asig poscil .5, (afrq*500)+1000,10 ;scale values of table 500 times, add 1000 Hz
outs asig , asig ;so we can distinguish the different tables

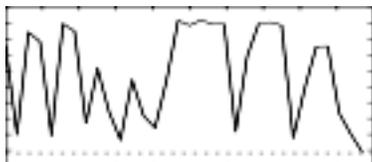
endin

</CsInstruments>
<CsScore>
f1 0 32 21 1 ;Uniform (white noise)
f2 0 32 21 6 ;Gaussian (mu=0.0, sigma=1/3.83=0.261)
f3 0 32 21 6 5.745 ;Gaussian (mu=0.0, sigma=5.745/3.83=1.5)
f4 0 32 21 9 1 1 2 ;Beta (note that level precedes arguments)
f5 0 32 21 10 1 2 ;Weibull
f10 0 8192 10 1 ;Sine wave

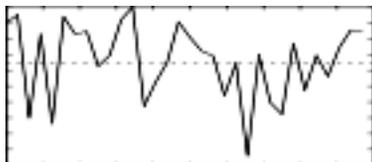
i 1 0 10 1
i 1 11 10 2
i 1 22 10 3
i 1 33 10 4
i 1 44 10 5
```

```
e  
</CsScore>  
</CsoundSynthesizer>
```

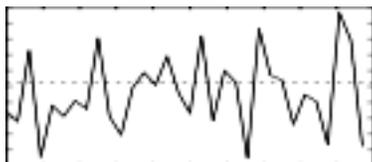
These are the diagrams of the waveforms of the GEN21 routines, as used in the example:



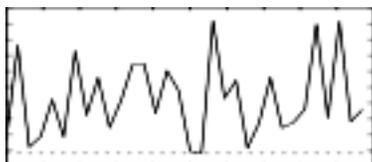
f 1 0 32 21 1 - positive numbers only



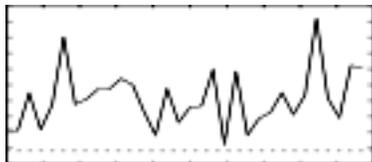
f 2 0 32 21 6



f 3 0 32 21 6 5.745



f 4 0 32 21 9 1 1 2 - positive numbers only



f 5 0 32 21 10 1 2 - positive numbers only

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK

Precisions about mu and sigma added by François Pinot after a discussion with Joachim Heintz on the Csound List, December 2010.

New in Csound version 3.2

# GEN23

GEN23 — Reads numeric values from a text file.

## Description

This subroutine reads numeric values from an external ASCII file.

## Syntax

```
f # time size -23 "filename.txt"
```

## Initialization

*"filename.txt"* -- numeric values contained in "filename.txt" (which indicates the complete pathname of the character file to be read), can be separated by spaces, tabs, newline characters or commas.

*size* -- number of points in the table. Must be a power of 2, power of 2 + 1, or zero. If *size* = 0, table size is determined by the number of numeric values in *filename.txt*. (New in Csound version 3.57)



### Note

All characters following ';', '#' (comment) or '<' (XML tag from version 6.04) are ignored until next line (numbers too).

## Examples

Here is an example of the GEN23 generator. It uses the file *gen23.csd* [examples/gen23.csd] and *spectrum.txt* [examples/spectrum.txt].

### Example 1200. Example of the GEN23 generator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gen23.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
;"spectrum.txt" is created by the spectrum plotter of Audacity (set at size 128), using "fox.wav".

instr 1 ;performs additive synthesis based on spectrum.txt

indx = 0      ;start reading at first value
```

```

loop:
ifreq tab_i indx, 2 ;take odd values of list (= frequency)
iamp tab_i indx+1, 2 ;take even values of list (= amplitude)
event_i "i", 10, 0, p3, iamp, ifreq ;use "event_i" to trigger instr. 10
    loop_lt indx, 2, 126, loop ;use all 128 frequency and amplitude values

endin

instr 10 ;generate sound

iamp = p4
ifreq = p5
asig poscil ampdb(iamp), ifreq, 1
asig linen asig, .01, p3, p2
outs asig, asig

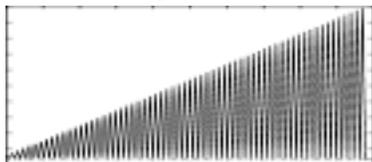
endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine wave
f 2 0 128 -23 "spectrum.txt" ;"spectrum.txt" can be found in /manual/examples

i1 0 2

e
</CsScore>
</CsoundSynthesizer>

```

This is the diagram of the waveform of the GEN23 routine, as used in the example:



f 2 0 128 -23 "spectrum.txt" - not normalized

## Credits

Author: Gabriel Maldonado  
Italy  
February, 1998

New in Csound version 3.47. Comments starting with '#' are ignored from Csound version 5.12.

# GEN24

GEN24 — Reads numeric values from another allocated function-table and rescales them.

## Description

This subroutine reads numeric values from another allocated function-table and rescales them according to the max and min values given by the user.

## Syntax

```
f # time size -24 ftable min max
```

## Initialization

*#, time, size* -- the usual GEN parameters. See *f* statement.

*ftable* -- *ftable* must be an already allocated table with the same size as this function.

*min, max* -- the rescaling range.



### Note

This GEN is useful, for example, to eliminate the starting offset in exponential segments allowing a real starting from zero.

## Examples

Here is an example of the GEN24 opcode. It uses the file *gen24.csd* [examples/gen24.csd].

### Example 1201. Example of the GEN24 generator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gen24.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ifn = p4      ;choose between tables
kcps init 1/p3 ;create index over duration of note.
```

```
kndx phasor kcps
ixmode = 1 ;normalize to 0-1
kval table kndx, ifn, ixmode
asig poscil .7, 440 + kval, 1 ;add to frequency
    outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine wave
f 10 0 16384 -24 1 0 400;scale sine wave from table 1 from 0 to 400
f 11 0 16384 -24 1 0 50 ;and from 0 to 50

i 1 0 3 10
i 1 4 3 11
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Gabriel Maldonado

New in Csound version 4.16

# GEN25

GEN25 — Construct functions from segments of exponential curves in breakpoint fashion.

## Description

These subroutines are used to construct functions from segments of exponential curves in breakpoint fashion.

## Syntax

```
f # time size 25 x1 y1 x2 y2 x3 ...
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see *f statement*).

*x1*, *x2*, *x3*, etc. -- locations in table at which to attain the following *y* value. Must be in increasing order. If the last value is less than *size*, then the rest will be set to zero. Should not be negative but can be zero.

*y1*, *y2*, *y3*, etc. -- Breakpoint values attained at the location specified by the preceding *x* value. These must be non-zero and must be alike in sign.



### Note

If *p4* is positive, functions are post-normalized (rescaled to a maximum absolute value of 1 after generation). A negative *p4* will cause rescaling to be skipped.

## Examples

Here is an example of the GEN25 generator. It uses the file *gen25.csd* [examples/gen25.csd].

### Example 1202. Example of the GEN25 generator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o gen25.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gisin ftgen 1, 0, 32768, 10, 1
gienv ftgen 2, 0, 1025, 25, 0, 0.01, 200, 1, 400, 1, 513, 0.01 ; y value must be >= 0
```

```

instr 1

kcps init 3/p3 ;play 3x over duration of note
kndx phasor kcps
ixmode = 1 ;normalize to 0-1
kval table kndx, gienv, ixmode
kval =kval*100 ;scale up to 0-100
asig poscil 1, 220+kval, gisin ;use table for amplitude
outs asig, asig

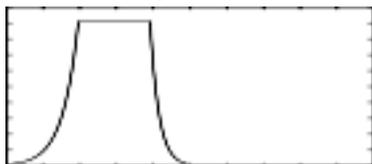
endin
</CsInstruments>
<CsScore>

i 1 0 4

e
</CsScore>
</CsoundSynthesizer>

```

This is the diagram of the waveform of the GEN25 routine, as used in the example:



f 2 0 1025 25 0 0.01 200 1 400 1 513 0.01 - a function which begins at 0.01, rises to 1 at the 200th table location, makes a straight line to the 400th location, and returns to 0.01 by the end of the table

## See Also

*f* statement, GEN27

## Credits

Author: John ffitch  
 University of Bath/Codemist Ltd.  
 Bath, UK

New in Csound version 3.49

# GEN27

GEN27 — Construct functions from segments of straight lines in breakpoint fashion.

## Description

Construct functions from segments of straight lines in breakpoint fashion.

## Syntax

```
f # time size 27 x1 y1 x2 y2 x3 ...
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see *f statement*).

*x1*, *x2*, *x3*, etc. -- locations in table at which to attain the following *y* value. Must be in increasing order. If the last value is less than *size*, then the rest will be set to zero. Should not be negative but can be zero.

*y1*, *y2*, *y3*, etc. -- Breakpoint values attained at the location specified by the preceding *x* value.



### Note

If *p4* is positive, functions are post-normalized (rescaled to a maximum absolute value of 1 after generation). A negative *p4* will cause rescaling to be skipped.

## Examples

Here is an example of the GEN27 generator. It uses the file *gen27.csd* [examples/gen27.csd].

### Example 1203. Example of the GEN27 generator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gen27.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

gisin ftgen 1, 0, 32768, 10, 1
gienv ftgen 2, 0, 1025, 27, 0, 0,200, 1, 400, -1, 513, 0

instr 1
```

```

kcps init 3/p3 ;play 3x over duration of note
kndx phasor kcps
ixmode = 1 ;normalize to 0-1
kval table kndx, gienv, ixmode
kval = kval*100 ;scale 0-100
asig poscil 1, 220+kval, 1 ;add to 220 Hz
      outs asig, asig

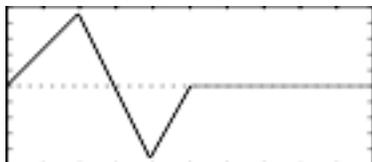
endin
</CsInstruments>
<CsScore>

i 1 0 4

e
</CsScore>
</CsoundSynthesizer>

```

This is the diagram of the waveform of the GEN27 routine, as used in the example:



f 2 0 1025 27 0 0 200 1 400 -1 513 0 - a function which begins at 0, rises to 1 at the 200th table location, falls to -1, by the 400th location, and returns to 0 by the end of the table. The interpolation is linear

## See Also

*f* statement, GEN25

## Credits

Author: John ffitch  
 University of Bath/Codemist Ltd.  
 Bath, UK

New in Csound version 3.49

# GEN28

GEN28 — Reads a text file which contains a time-tagged trajectory.

## Description

This function generator reads a text file which contains sets of three values representing the xy coordinates and a time-tag for when the signal should be placed at that location, allowing the user to define a time-tagged trajectory. The file format is in the form:

```
time1 X1 Y1
time2 X2 Y2
time3 X3 Y3
```

The configuration of the xy coordinates in space places the signal in the following way:

- a1 is -1, 1
- a2 is 1, 1
- a3 is -1, -1
- a4 is 1, -1

This assumes a loudspeaker set up as a1 is left front, a2 is right front, a3 is left back, a4 is right back. Values greater than 1 will result in sounds being attenuated as if in the distance. *GEN28* creates values to 10 milliseconds of resolution.

## Syntax

```
f # time size 28 ifilcod
```

## Initialization

*size* -- number of points in the table. Must be 0. *GEN28* takes 0 as the size and automatically allocates memory.

*ifilcod* -- character-string denoting the source file name. A character-string (in double quotes, spaces permitted) gives the filename itself, optionally a full pathname. If not a full path, the named file is sought in the current directory.

## Examples

```
f1 0 0 28 "move"
```

The file "move" should look like:

```
0 -1 1
1 1 1
```

2	4	4
2.1	-4	-4
3	10	-10
5	-40	0

Since *GEN28* creates values to 10 milliseconds of resolution, there will be 500 values created by interpolating X1 to X2 to X3 and so on, and Y1 to Y2 to Y3 and so on, over the appropriate number of values that are stored in the function table. The sound will begin in the left front, over 1 second it will move to the right front, over another second it move further into the distance but still in the right front, then in just 1/10th of a second it moves to the left rear, a bit distant. Finally over the last .9 seconds the sound will move to the right rear, moderately distant, and it comes to rest between the two left channels (due west!), quite distant.

Here is an example of the GEN28 routine. It uses the file *gen28.csd* [examples/gen28.csd].

### Example 1204. Example of the gen28 generator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gen28.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
Odbfs = 1
nchnls = 4

ga1 init 0
ga2 init 0
ga3 init 0
ga4 init 0

instr 1 ;uses GEN28 file "move", as found in /manual/examples

kx  init 0
ky  init 0
ktime line 0, 5, 5 ;same time as in table 1 ("move")
asig diskint2 "beats.wav", 1, 0, 1 ;sound source is looped
a1, a2, a3, a4 space asig, 1, ktime, .1, kx, ky ;use table 1 = GEN28
ar1, ar2, ar3, ar4 spsend ;send to reverb

ga1 = ga1+ar1
ga2 = ga2+ar2
ga3 = ga3+ar3
ga4 = ga4+ar4
outq a1, a2, a3, a4

endin

instr 99 ; reverb instrument

a1 reverb2 ga1, 2.5, .5
a2 reverb2 ga2, 2.5, .5
a3 reverb2 ga3, 2.5, .5

```

```
a4 reverb2 ga4, 2.5, .5
  outq a1, a2, a3, a4

ga1=0
ga2=0
ga3=0
ga4=0

endin
</CsInstruments>
<CsScore>
f1 0 0 28 "move"

i1 0 5 ;same time as ktime
i 99 0 10 ;keep reverb active
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Richard Karpen  
Seattle, Wash  
1998

New in Csound version 3.48

# GEN30

GEN30 — Generates harmonic partials by analyzing an existing table.

## Description

Extracts a range of harmonic partials from an existing waveform.

## Syntax

```
f # time size 30 src minh maxh [ref_sr] [interp]
```

## Performance

*src* -- source ftable

*minh* -- lowest harmonic number

*maxh* -- highest harmonic number

*ref\_sr* (optional) -- *maxh* is scaled by (*sr* / *ref\_sr*). The default value of *ref\_sr* is *sr*. If *ref\_sr* is zero or negative, it is now ignored.

*interp* (optional) -- if non-zero, allows changing the amplitude of the lowest and highest harmonic partial depending on the fractional part of *minh* and *maxh*. For example, if *maxh* is 11.3 then the 12th harmonic partial is added with 0.3 amplitude. This parameter is zero by default.

*GEN30* does not support tables with an extended guard point (ie. table size = power of two + 1). Although such tables will work both for input and output, when reading source table(s), the guard point is ignored, and when writing the output table, guard point is simply copied from the first sample (table index = 0).

The reason of this limitation is that *GEN30* uses FFT, which requires power of two table size. *GEN32* allows using linear interpolation for resampling and phase shifting, which makes it possible to use any table size (however, for partials calculated with FFT, the power of two limitation still exists).

## Examples

Here is an example of the GEN30 routine. It uses the files *gen30.csd* [examples/gen30.csd].

### Example 1205. Example of the GEN30 routine.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      ;;realtime audio out
; For Non-realtime ouput leave only the line below:
; -o gen30.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```

;a simplified example of Istvan Varga
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

isaw ftgen 1, 0, 16384, 7, 1, 16384, -1 ;sawtooth wave
iFM ftgen 3, 0, 4096, 7, 0, 512, 0.25, 512, 1, 512, 0.25, 512, \
    0, 512, -0.25, 512, -1, 512, -0.25, 512, 0 ;FM waveform
iAM ftgen 4, 0, 4096, 5, 1, 4096, 0.01 ;AM waveform
iEQ ftgen 5, 0, 1024, 5, 1, 512, 32, 512, 1 ;FM to EQ
isine ftgen 6, 0, 1024, 10, 1 ;sine wave

/* generate bandlimited sawtooth waves */
i0 = 0
loop1:
imaxh = sr / (2 * 440.0 * exp(log(2.0) * (i0 - 69) / 12))
i_ ftgen i0 + 10, 0, 4096, -30, 1, 1, imaxh ;use gen 30
i0 = i0 + 1
if (i0 < 127.5) igoto loop1

instr 1

kcps = 440.0 * exp(log(2.0) * (p4 - 69) / 12) ;note frequency
klpmaxf limit p5 * kcps, 1000.0, 12000.0 ;lowpass max. frequency

kfmd1 = 0.03 * kcps ;FM depth in Hz
kamfr = kcps * 0.02 ;AM frequency
kamfr2 = kcps * 0.1

kfnum = (10 + 69 + 0.5 + 12 * log(kcps / 440.0) / log(2.0)) ;table number
aenv linseg 0, p3*0.25, 1, p3*0.75, 0 ;amp. envelope

asig oscbnk kcps, 0.0, kfmd1, 0.0, 40, 200, 0.1, 0.2, 0, 0, 144, \
    0.0, klpmaxf, 0.0, 0.0, 1.5, 1.5, 2, kfnum, 3, 0, 5, 5, 5
asig = asig * aenv*.03
outs asig, asig

endin

</CsInstruments>
<CsScore>
s
i 1 0 6 41 10
i 1 0 6 60
i 1 0 6 65
i 1 0 6 69
s
i 1 0 6 41 64
i 1 0 6 60
i 1 0 6 65
i 1 0 6 69
e
</CsScore>
</CsSoundSynthesizer>

```

## Credits

Author: Istvan Varga

New in version 4.16

# GEN31

GEN31 — Mixes any waveform specified in an existing table.

## Description

This routine is similar to GEN09, but allows mixing any waveform specified in an existing table.

## Syntax

```
f # time size 31 src pna stra phsa pnb strb phsb ...
```

## Performance

*src* -- source table number

*pna, pnb, ...* -- partial number, must be a positive integer

*stra, strb, ...* -- amplitude scale

*phsa, phsb, ...* -- start phase (0 to 1)

*GEN31* does not support tables with an extended guard point (ie. table size = power of two + 1). Although such tables will work both for input and output, when reading source table(s), the guard point is ignored, and when writing the output table, guard point is simply copied from the first sample (table index = 0).

The reason of this limitation is that *GEN31* uses FFT, which requires power of two table size. *GEN32* allows using linear interpolation for resampling and phase shifting, which makes it possible to use any table size (however, for partials calculated with FFT, the power of two limitation still exists).

## Credits

Author: Istvan Varga

New in version 4.15

# GEN32

GEN32 — Mixes any waveform, resampled with either FFT or linear interpolation.

## Description

This routine is similar to *GEN31*, but allows specifying source ftable for each partial. Tables can be resampled either with FFT, or linear interpolation.

## Syntax

```
f # time size 32 srca pna stra phsa srcb pnb strb phsb ...
```

## Performance

*srca*, *srcb* -- source table number. A negative value can be used to read the table with linear interpolation (by default, the source waveform is transposed and phase shifted using FFT); this is less accurate, but faster, and allows non-integer and negative partial numbers.

*pna*, *pnb*, ... -- partial number, must be a positive integer if source table number is positive (i.e. resample with FFT).

*stra*, *strb*, ... -- amplitude scale

*phsa*, *phsb*, ... -- start phase (0 to 1)

## Examples

Here is an example of the GEN32 routine. It uses the file *gen32.csd* [examples/gen32.csd].

### Example 1206. Example of the gen32 generator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;;realtime audio out
;-iadc     ;;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gen32.wav -W ;;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

itmp   ftgen 1, 0, 16384, 7, 1, 16384, -1 ; sawtooth
itmp   ftgen 2, 0, 8192, 10, 1 ; sine
itmp   ftgen 5, 0, 4096, -32, -2, 1.5, 1.0, 0.25, 1, 2, 0.5, 0, 1, 3, -0.25, 0.5 ; mix tables
itmp   ftgen 6, 0, 16384, 20, 3, 1 ; window
```

```

; generate band-limited waveforms
inote = 0
loop0:
icps = 440 * exp(log(2) * (inote - 69) / 12) ; one table for
inumh = sr / (2 * icps) ; each MIDI note number
ift = int(inote + 256.5)
itmp ftgen ift, 0, 4096, -30, 5, 1, inumh
inote = inote + 1
if (inote < 127.5) igoto loop0

instr 1

kcps expon 20, p3, 16000
kft = int(256.5 + 69 + 12 * log(kcps / 440) / log(2))
kft = (kft > 383 ? 383 : kft)
a1 phasor kcps
a1 tableikt a1, kft, 1, 0, 1
outs a1*.5, a1*.5

endin

instr 2

kcps expon 20, p3, 16000
kft = int(256.5 + 69 + 12 * log(kcps / 440) / log(2))
kft = (kft > 383 ? 383 : kft)
kgdur limit 10 / kcps, 0.1, 1
a1 grain2 kcps, 0.02, kgdur, 30, kft, 6, -0.5
outs a1*.08, a1*.08

endin
</CsInstruments>
<CsScore>
t 0 60

i 1 0 10
i 2 12 10
e
</CsScore>
</CsoundSynthesizer>

```

## Credits

Author: Rasmus Ekman

Programmer: Istvan Varga

New in version 4.17

# GEN33

GEN33 — Generate composite waveforms by mixing simple sinusoids.

## Description

These routines generate composite waveforms by mixing simple sinusoids, similarly to *GEN09*, but the parameters of the partials are specified in an already existing table, which makes it possible to calculate any number of partials in the orchestra.

The difference between *GEN33* and *GEN34* is that *GEN33* uses inverse FFT to generate output, while *GEN34* is based on the algorithm used in *oscils* opcode. *GEN33* allows integer partials only, and does not support power of two plus 1 table size, but may be significantly faster with a large number of partials. On the other hand, with *GEN34*, it is possible to use non-integer partial numbers and extended guard point, and this routine may be faster if there is only a small number of partials (note that *GEN34* is also several times faster than *GEN09*, although the latter may be more accurate).

## Syntax

```
f # time size 33 src nh scl [fmode]
```

## Initialization

*size* -- number of points in the table. Must be power of two and at least 4.

*src* -- source table number. This table contains the parameters of each partial in the following format:

```
stra, pna, phsa, strb, pnb, phsb, ...
```

the parameters are:

- *stra*, *strb*, etc.: relative strength of partials. The actual amplitude depends on the value of *scl*, or normalization (if enabled).
- *pna*, *pnb*, etc.: partial number, or frequency, depending on *fmode* (see below); zero and negative values are allowed, however, if the absolute value of the partial number exceeds ( $size / 2$ ), the partial will not be rendered. With *GEN33*, partial number is rounded to the nearest integer.
- *phsa*, *phsb*, etc.: initial phase, in the range 0 to 1.

Table length (not including the guard point) should be at least  $3 * nh$ . If the table is too short, the number of partials (*nh*) is reduced to  $(table\ length) / 3$ , rounded towards zero.

*nh* -- number of partials. Zero or negative values are allowed, and result in an empty table (silence). The actual number may be reduced if the source table (*src*) is too short, or some partials have too high frequency.

*scl* -- amplitude scale.

*fmode* (optional, default = 0) -- a non-zero value can be used to set frequency in Hz instead of partial numbers in the source table. The sample rate is assumed to be *fmode* if it is positive, or  $-(sr * fmode)$  if any negative value is specified.

## Examples

Here is an example of the GEN33 routine. It uses the file *gen33.csd* [examples/gen33.csd].

### Example 1207. Example of the gen33 generator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gen33.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; partials 1, 4, 7, 10, 13, 16, etc. with base frequency of 400 Hz

ibsfrq = 400
inumh = int(1.5 + sr * 0.5 / (3 * ibsfrq)) ; estimate number of partials
isrcln = int(0.5 + exp(log(2) * int(1.01 + log(inumh * 3) / log(2)))) ; source table length
itmp ftgen 1, 0, isrcln, -2, 0 ; create empty source table
ifpos = 0
ifrq = ibsfrq
inumh = 0
l1:
tableiw ibsfrq / ifrq, ifpos, 1 ; amplitude
tableiw ifrq, ifpos + 1, 1 ; frequency
tableiw 0, ifpos + 2, 1 ; phase
ifpos = ifpos + 3
ifrq = ifrq + ibsfrq * 3
inumh = inumh + 1
if (ifrq < (sr * 0.5)) igoto l1

itemp ftgen 2, 0, 262144, -33, 1, inumh, 1, -1 ; store output in ftable 2 (size = 262144)
asig poscil .5, ibsfrq, itemp
outs asig, asig

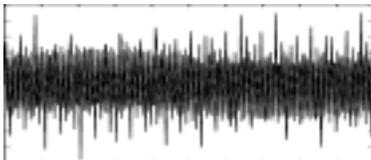
endin
</CsInstruments>
<CsScore>

i 1 0 2

e
</CsScore>
</CsoundSynthesizer>

```

This is the diagram of the waveform generated by GEN33, as used in the example:



f 2 0 262144 -33 1 inumh 1 -1

## See Also

*GEN09, GEN34*

## Credits

Programmer: Istvan Varga  
March 2002

New in version 4.19

# GEN34

GEN34 — Generate composite waveforms by mixing simple sinusoids.

## Description

These routines generate composite waveforms by mixing simple sinusoids, similarly to *GEN09*, but the parameters of the partials are specified in an already existing table, which makes it possible to calculate any number of partials in the orchestra.

The difference between *GEN33* and *GEN34* is that *GEN33* uses inverse FFT to generate output, while *GEN34* is based on the algorithm used in *oscils* opcode. *GEN33* allows integer partials only, and does not support power of two plus 1 table size, but may be significantly faster with a large number of partials. On the other hand, with *GEN34*, it is possible to use non-integer partial numbers and extended guard point, and this routine may be faster if there is only a small number of partials (note that *GEN34* is also several times faster than *GEN09*, although the latter may be more accurate).

## Syntax

```
f # time size 34 src nh scl [fmode]
```

## Initialization

*size* -- number of points in the table. Must be power of two or a power of two plus 1.

*src* -- source table number. This table contains the parameters of each partial in the following format:

```
stra, pna, phsa, strb, pnb, phsb, ...
```

the parameters are:

- *stra*, *strb*, etc.: relative strength of partials. The actual amplitude depends on the value of *scl*, or normalization (if enabled).
- *pna*, *pnb*, etc.: partial number, or frequency, depending on *fmode* (see below); zero and negative values are allowed, however, if the absolute value of the partial number exceeds ( $size / 2$ ), the partial will not be rendered.
- *phsa*, *phsb*, etc.: initial phase, in the range 0 to 1.

Table length (not including the guard point) should be at least  $3 * nh$ . If the table is too short, the number of partials (*nh*) is reduced to  $(table\ length) / 3$ , rounded towards zero.

*nh* -- number of partials. Zero or negative values are allowed, and result in an empty table (silence). The actual number may be reduced if the source table (*src*) is too short, or some partials have too high frequency.

*scl* -- amplitude scale.

*fmode* (optional, default = 0) -- a non-zero value can be used to set frequency in Hz instead of partial numbers in the source table. The sample rate is assumed to be *fmode* if it is positive, or  $-(sr * fmode)$  if any negative value is specified.

## Examples

Here is an example of the GEN34 routine. It uses the file *gen34.csd* [examples/gen34.csd].

### Example 1208. Example of the gen34 generator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gen34.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; partials 1, 4, 7, 10, 13, 16, etc. with base frequency of 400 Hz

ibsfrq = 400
inumh = int(1.5 + sr * 0.5 / (3 * ibsfrq)) ; estimate number of partials
isrcln = int(0.5 + exp(log(2) * int(1.01 + log(inumh * 3) / log(2)))) ; source table length
itmp ftgen 1, 0, isrcln, -2, 0 ; create empty source table
ifpos = 0
ifrq = ibsfrq
inumh = 0
l1:
tableiw ibsfrq / ifrq, ifpos, 1 ; amplitude
tableiw ifrq, ifpos + 1, 1 ; frequency
tableiw 0, ifpos + 2, 1 ; phase
ifpos = ifpos + 3
ifrq = ifrq + ibsfrq * 3
inumh = inumh + 1
if (ifrq < (sr * 0.5)) igoto l1

itmp ftgen 2, 0, 262144, -34, 1, inumh, 1, -1 ; store output in ftable 2 (size = 262144)
asig poscil .5, ibsfrq, itmp
outs asig, asig

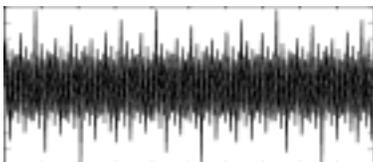
endin
</CsInstruments>
<CsScore>

i 1 0 2

e
</CsScore>
</CsoundSynthesizer>

```

This is the diagram of the waveform generated by GEN34, as used in the example:



f 2 0 262144 -34 1 inumh 1 -1

## See Also

*GEN09, GEN33*

## Credits

Programmer: Istvan Varga  
March 2002

New in version 4.19

# GEN40

GEN40 — Generates a random distribution using a distribution histogram.

## Description

Generates a continuous random distribution function starting from the shape of a user-defined distribution histogram.

## Syntax

```
f # time size 40 shapetab
```

## Performance

The shape of histogram must be stored in a previously defined table, in fact shapetab argument must be filled with the number of such table.

Histogram shape can be generated with any other GEN routines. Since no interpolation is used when GEN40 processes the translation, it is suggested that the size of the table containing the histogram shape to be reasonably big, in order to obtain more precision (however after the processing the shaping-table can be destroyed in order to re-gain memory).

This subroutine is designed to be used together with cusernd opcode (see cusernd for more information).

## Examples

Here is an example of the GEN40 generator. It uses the file *gen40.csd* [examples/gen40.csd].

### Example 1209. Example of the GEN40 generator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o GEN40.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; every run time same values

kuser cusernd 0, 100, 1
printk .2, kuser
asig poscil .5, 220+kuser, 3
outs asig, asig
endin
```

```
instr 2 ; every run time different values

    seed 0
kuser cuserrnd 0, 100, 1
    printk .2, kuser
asig poscil .5, 220+kuser, 3
    outs asig, asig
endin
</CsInstruments>
<CsScore>
f 1 0 16 -7 1 4 0 8 0 4 1 ;distrubution using GEN07
f 2 0 16384 40 1 ;GEN40 is to be used with cuserrnd
f 3 0 8192 10 1 ;sine

i 1 0 2
i 2 3 2
e
</CsScore>
</CsoundSynthesizer>
```

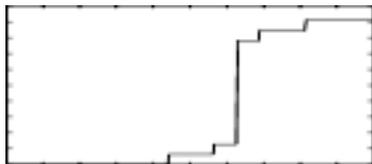
Its output should include lines like these:

```
i 1 time      0.00067:    53.14918
i 1 time      0.20067:     0.00000
i 1 time      0.40067:     0.00000
i 1 time      0.60067:    96.80406
i 1 time      0.80067:    94.20729
i 1 time      1.00000:     0.00000
i 1 time      1.20067:    86.13032
i 1 time      1.40067:    31.37096
i 1 time      1.60067:    70.35889
i 1 time      1.80000:     0.00000
i 1 time      2.00000:    49.18914
```

WARNING: Seeding from current time 2006647442

```
i 2 time      3.00067:    21.45002
i 2 time      3.20067:    44.32333
i 2 time      3.40067:    46.05420
i 2 time      3.60000:     0.00000
i 2 time      3.80067:    41.32175
i 2 time      4.00000:     0.00000
i 2 time      4.20000:    63.72019
i 2 time      4.40067:     0.00000
i 2 time      4.60067:     0.00000
i 2 time      4.80067:     0.00000
i 2 time      5.00000:    74.49330
```

This is the diagram of the waveform of the GEN40 routine, as used in the example:



```
f 2 0 16384 40 1
```

## Credits

Author: Gabriel Maldonado

# GEN41

GEN41 — Generates a random list of numerical pairs.

## Description

Generates a discrete random distribution function by giving a list of numerical pairs.

## Syntax

```
f # time size -41 value1 prob1 value2 prob2 value3 prob3 ... valueN probN
```

## Performance

The first number of each pair is a value, and the second is the probability of that value to be chosen by a random algorithm. Even if any number can be assigned to the probability element of each pair, it is suggested to give it a percent value, in order to make it clearer for the user.

This subroutine is designed to be used together with `duserrnd` and `urd` opcodes (see `duserrnd` for more information).

## Examples

Here is an example of the GEN41 generator. It uses the file `gen41.csd` [examples/gen41.csd].

### Example 1210. Example of the GEN41 generator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if RT audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o GEN41.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

k1  duserrnd 1
    printk 0, k1
asig poscil .5, 220*k1, 2
    outs asig, asig

endin
</CsInstruments>
<CsScore>
```

```
f1 0 -20 -41 2 .1 8 .9 ;choose 2 at 10% probability, 8 at 90%  
f2 0 8192 10 1  
i1 0 2  
e  
</CsScore>  
</CsoundSynthesizer>
```

Its output should include lines like these:

```
i 1 time 0.00067: 8.00000  
i 1 time 0.00133: 8.00000  
i 1 time 0.00200: 8.00000  
i 1 time 0.00267: 8.00000  
i 1 time 0.00333: 2.00000  
i 1 time 0.00400: 8.00000  
i 1 time 0.00533: 8.00000  
i 1 time 0.00600: 8.00000  
.....
```

This is the diagram of the waveform of the GEN41 routine, as used in the example:



```
f 1 0 -20 -41 2 .1 8 .9
```

## Credits

Author: Gabriel Maldonado

# GEN42

GEN42 — Generates a random distribution of discrete ranges of values.

## Description

Generates a random distribution function of discrete ranges of values by giving a list of groups of three numbers.

## Syntax

```
f # time size -42 min1 max1 prob1 min2 max2 prob2 min3 max3 prob3 ... minN maxN probN
```

## Performance

The first number of each group is the minimum value of the range, the second is the maximum value and the third is the probability of that an element belonging to that range of values can be chosen by a random algorithm. Probabilities for a range should be a fraction of 1, and the sum of the probabilities for all the ranges should total 1.0.

This subroutine is designed to be used together with `dusernd` and `urd` opcodes (see `dusernd` for more information). Since both `dusernd` and `urd` do not use any interpolation, it is suggested to give a size reasonably big.

## Examples

Here is an example of the GEN42 generator. It uses the file `gen42.csd` [examples/gen42.csd].

### Example 1211. Example of the GEN42 generator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac    ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o GEN42.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

ktab = 1 ;ftable 1
kurd = urd(ktab)
ktrig metro 5 ;triggers 5 times per second
kres samphold kurd, ktrig ;sample and hold value of kurd
```

```

    printk2 kres ;print it
asig poscil .5, 220+kres, 2
outs asig, asig
endin

instr 2

seed 0 ;every run new values

ktab = 1 ;ftable 1
kurd = urd(ktab)
ktrig metro 5 ;triggers 5 times per second
kres samphold kurd, ktrig ;sample and hold value of kurd
printk2 kres ;print it
asig poscil .5, 220+kres, 2
outs asig, asig

endin
</CsInstruments>
<CsScore>
f1 0 -20 -42 10 20 .3 100 200 .7 ;30% choose between 10 and 20 and 70% between 100 and 200
f2 0 8192 10 1 ;sine wave

i 1 0 5
i 2 6 5
e
</CsScore>
</CsoundSynthesizer>

```

Its output should include lines like these:

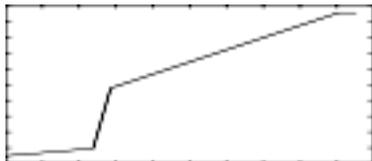
```

i1 184.61538
i1 130.76923
i1 169.23077
i1 12.00000
.....
WARNING: Seeding from current time 3751086165

i2 138.46154
i2 12.00000
i2 123.07692
i2 161.53846
i2 123.07692
i2 153.84615
.....

```

This is the diagram of the waveform of the GEN42 routine, as used in the example:



f 1 0 -20 -42 10 20 .3 100 200 .7

## Credits

Author: Gabriel Maldonado

# GEN43

GEN43 — Loads a PVOCEX file containing a PV analysis.

## Description

This subroutine loads a PVOCEX file containing the PV analysis (amp-freq) of a soundfile and calculates the average magnitudes of all analysis frames of one or all audio channels. It then creates a table with these magnitudes for each PV bin.

## Syntax

```
f # time size 43 filecod channel
```

## Initialisation

*size* -- number of points in the table, power-of-two or power-of-two plus 1. GEN 43 does not make any distinction between these two sizes, but it requires the table to be at least the `fftsize/2`. PV bins cover the positive spectrum from 0Hz (table index 0) to the Nyquist (table index `fftsize/2+1`) in equal-size frequency increments (of size `sr/fftsize`).

*filecod* -- a pvocex file (which can be generated by `pvanal`).

*channel* -- audio channel number from which the magnitudes will be extracted; a 0 will average the magnitudes from all channels.

Reading stops at the end of the file.



### Note

if `p4` is positive, the table will be post-normalised. A negative `p4` will cause post-normalisation to be skipped.

## Examples

```
f1 0 512 43 "viola.pvx" 1
f1 0 -1024 -43 "noiseprint.pvx" 0
```

This table can be used as a masking table for `pvstencil` and `pvsmaska`. The first example uses a 1024-point FFT phase vocoder analysis file from which the first channel is used. The second uses all channels of a 2048-point file, without post-normalisation. For noise reduction applications with `pvstencil`, it is easiest to skip table normalisation (negative GEN code).

## Credits

Author: Victor Lazzarini

# GEN49

GEN49 — Transfers data from an MP3 soundfile into a function table.

## Description

This subroutine transfers data from an MP3 soundfile into a function table.

## Syntax

```
f# time size 49 filcod skiptime format
```

## Performance

*size* -- number of points in the table. Ordinarily a power of 2 or a power-of-2 plus 1 (see *f statement*); the maximum table size is 16777216 ( $2^{24}$ ) points. The allocation of table memory can be *deferred* by setting this parameter to 0; the size allocated is then the number of points in the file (probably not a power-of-2), and the table is not usable by normal oscillators, but it is usable by a *loscil* unit. The soundfile can also be mono or stereo.

*filcod* -- integer or character-string denoting the source soundfile name. An integer denotes the file *soundin.filcod*; a character-string (in double quotes, spaces permitted) gives the filename itself, optionally a full pathname. If not a full path, the file is sought first in the current directory, then in that given by the environment variable *SSDIR* (if defined) then by *SFDIR*. See also *soundin*.

*skiptime* -- begin reading at *skiptime* seconds into the file.

*format* -- specifies the audio data-file format required:

1 - Mono file	3 - First channel (left)
2 - Stereo file	4 - Second channel (right)

If *format* = 0 the sample format is taken from the soundfile header.



### Note

- Reading stops at end-of-file or when the table is full. Table locations not filled will contain zeros.
- If *p4* is positive, the table will be post-normalized (rescaled to a maximum absolute value of 1 after generation). A negative *p4* will cause rescaling to be skipped.

## Examples

Here is an example of the GEN49 routine. It uses the files *gen49.csd* [examples/gen49.csd].

**Example 1212. An example of the GEN49 routine.**

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gen49.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

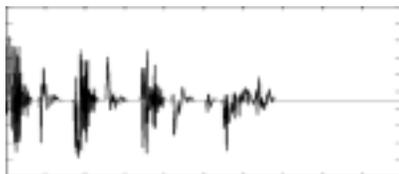
kcps = sr/ftlen(1)
asig oscil .8, kcps, 1
    outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 131072 49 "beats.mp3" 0 1 ;read an audio file (using GEN49).

i 1 0 2
e
</CsScore>
</CsoundSynthesizer>

```

This is the diagram of the waveforms of the GEN49 routine, as used in the example:



```
f 1 0 131072 49 "beats.mp3" 0 1
```

## Credits

Written by John ffitch

February 2009.

# GEN51

GEN51 — This subroutine fills a table with a fully customized micro-tuning scale, in the manner of Csound opcodes *cpstun*, *cpstuni* and *cpstmid*.

## Description

This subroutine fills a table with a fully customized micro-tuning scale, in the manner of Csound opcodes *cpstun*, *cpstuni* et *cpstmid*.

## Syntax

```
f # time size -51 numgrades interval basefreq basekey tuningRatio1 tuningRatio2 ... tuningRatioN
```

## Performance

The first four parameters (i.e. p5, p6, p7 and p8) define the following generation directives:

*p5 (numgrades)* -- the number of grades of the micro-tuning scale

*p6 (interval)* -- the frequency range covered before repeating the grade ratios, for example 2 for one octave, 1.5 for a fifth etcetera

*p7 (basefreq)* -- the base frequency of the scale in cps

*p8 (basekey)* -- the integer index of the scale to which to assign basefreq unmodified

The other parameters define the ratios of the scale:

*p9...pN (tuningRatio1...etc.)* -- the tuning ratios of the scale

For example, for a standard 12-grade scale with the base-frequency of 261 cps assigned to the key-number 60, the corresponding f-statement in the score to generate the table should be:

```
;          numgrades      basefreq      tuning-ratios (eq.temp) .....
;          interval      basekey
f1 0 64 -51      12          2          261      60          1  1.059463 1.12246 1.18920 ..etc...
```

After the gen has been processed, the table f1 is filled with 64 different frequency values. The 60th element is filled with the frequency value of 261, and all other elements (preceding and subsequents) of the table are filled according to the tuning ratios

Another example with a 24-grade scale with a base frequency of 440 assigned to the key-number 48, and a repetition interval of 1.5:

```
;          numgrades      basefreq      tuning-ratios .....
;          interval      basekey
f1 0 64 -51      24          1.5        440      48          1  1.01  1.02  1.03 ..etc...
```

## Examples

Here is an example of the GEN51 routine. It uses the files *gen51.csd* [examples/gen51.csd].

**Example 1213. An example of the GEN51 routine.**

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac -M0 ;;realtime audio out and midi input
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gen51.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
;example by Iain McCurdy
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giEqTmp12 ftgen 1,0,128,-51,12,2,cpsoct(8),60,1,2^(1/12),2^(2/12),2^(3/12),2^(4/12),2^(5/12),2^(6/12),2^(7/12),2^(8/12),2^(9/12),2^(10/12),2^(11/12)
giEqTmp10 ftgen 2,0,128,-51,10,2,cpsoct(8),60,1,2^(1/10),2^(2/10),2^(3/10),2^(4/10),2^(5/10),2^(6/10),2^(7/10),2^(8/10),2^(9/10),2^(10/10)
giEqTmp24 ftgen 3,0,128,-51,24,2,cpsoct(8),60,1,2^(1/24),2^(2/24),2^(3/24),2^(4/24),2^(5/24),2^(6/24),2^(7/24),2^(8/24),2^(9/24),2^(10/24),2^(11/24),2^(12/24),2^(13/24),2^(14/24),2^(15/24),2^(16/24),2^(17/24),2^(18/24),2^(19/24),2^(20/24),2^(21/24),2^(22/24),2^(23/24)

instr 1 ;midi input instrument
/*USE PITCH BEND TO MODULATE NOTE NUMBER UP OR DOWN ONE STEP - ACTUAL INTERVAL IT WILL MODULATE BY WILL BE 1/12 OF AN OCTAVE
;kbend pchbend 0,2

/*ALTERNATIVELY IF USING VIRTUAL MIDI DEVICE OR A KEYBOARD WITH NO PITCH BEND WHEEL, USE CONTROLLERS
kup ctrl7 1, 1, 0, 1
kdown ctrl7 1, 2, 0, -1
kbend = kup+kdown

inum notnum
kcps tablei inum+kbend, giEqTmp24 ;read cps values from GEN51, scale table using a combination of note
a1 vco2 0.2, kcps, 4, 0.5
outs a1, a1
endin

instr 2 ;score input instrument

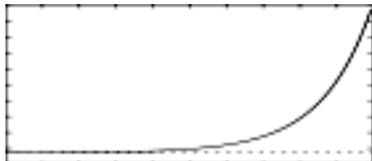
knum line p4, p3, p5 ;gliss using a straight line bewteen p4 and p5 for the entire note duration
kcps tablei knum, giEqTmp24 ;read cps values from GEN51 scale table
a1 vco2 0.2, kcps, 4, 0.5
outs a1, a1
endin

</CsInstruments>
<CsScore>
f 0 3600

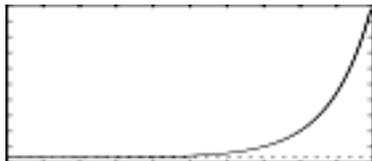
;instr 2. Score input. Gliss from step number p4 to step number p5
;p4 - starting note number
;p5 - ending note number
i 2 0 2 60 61
i 2 + 2 70 58
i 2 + 2 66 66.5
i 2 + 2 71.25 71
e
</CsScore>
</CsoundSynthesizer>

```

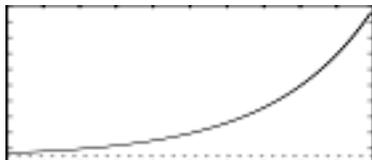
These are the diagrams of the waveforms of the GEN51 routines, as used in the example:



f 1 0 128 -51 12 2 cpsoct(8) 60 1  $2^{(1/12)}$   $2^{(2/12)}$   $2^{(3/12)}$   $2^{(4/12)}$   $2^{(5/12)}$   $2^{(6/12)}$   $2^{(7/12)}$   $2^{(8/12)}$   
 $2^{(9/12)}$   $2^{(10/12)}$   $2^{(11/12)}$   $2^{(12/12)}$



f 2 0 128 -51 10 2 cpsoct(8) 60 1  $2^{(1/10)}$   $2^{(2/10)}$   $2^{(3/10)}$   $2^{(4/10)}$   $2^{(5/10)}$   $2^{(6/10)}$   $2^{(7/10)}$   $2^{(8/10)}$   
 $2^{(9/10)}$   $2^{(10/10)}$



f 3 0 128 -51 24 2 cpsoct(8) 60 1  $2^{(1/24)}$   $2^{(2/24)}$   $2^{(3/24)}$   $2^{(4/24)}$   $2^{(5/24)}$   $2^{(6/24)}$   $2^{(7/24)}$   $2^{(8/24)}$   
 $2^{(9/24)}$   $2^{(10/24)}$   $2^{(11/24)}$   $2^{(12/24)}$   $2^{(13/24)}$   $2^{(14/24)}$   $2^{(15/24)}$   $2^{(16/24)}$   $2^{(17/24)}$   $2^{(18/24)}$   
 $2^{(19/24)}$   $2^{(20/24)}$   $2^{(21/24)}$   $2^{(22/24)}$   $2^{(23/24)}$   $2^{(24/24)}$

## Credits

Author: Gabriel Maldonado

# GEN52

GEN52 — Creates an interleaved multichannel table from the specified source tables, in the format expected by the *ftconv* opcode.

## Description

GEN52 creates an interleaved multichannel table from the specified source tables, in the format expected by the *ftconv* opcode. It can also be used to extract a channel from a multichannel table and store it in a normal mono table, copy tables with skipping some samples, adding delay, or store in reverse order, etc.

Three parameters must be given for each channel to be processed. *fsrc* declares the source f-table number. The parameter *offset* specifies an offset for the source file. If different to 0, the source file is not read from the beginning, but the *offset* number of values are skipped. The *offset* is used to determine the channel number to be read from interleaved f-tables, e.g. for channel 2, *offset* must be 1. It can also be used to set a read offset on the source table. This parameter gives absolute values, so if a skip of 20 sample frames for a 2 channel f-table is desired, *offset* must be set to 40. The *srcchnls* parameter is used to declare the number of channels in the source f-table. This parameter sets the skip size when reading the source f-table.

When more than one channel (*nchannels* > 1) is given, source f-tables are interleaved in the newly created table.

If the source f-table is finished before the destination f-table is full, the remaining values are set to 0.

## Syntax

```
f # time size 52 nchannels fsrc1 offset1 srcchnls1 [fsrc2 offset2 srcchnls2 ... fsrcN offsetN srcchnlsN]
```

## Example

```
; source tables
f 1 0 16384 10 1
f 2 0 16384 10 0 1
; create 2 channel interleaved table
f 3 0 32768 -52 2 1 0 1 2 0 1
; extract first channel from table 3
f 4 0 16384 -52 1 3 0 2
; extract second channel from table 3
f 5 0 16384 -52 1 3 1 2
```

Here is a complete example of the GEN52 generator. It uses the file *gen52.csd* [examples/gen52.csd].

### Example 1214. Example of the GEN52 generator.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac ;;realtime audio out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime output leave only the line below:
; -o gen52.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
```

```

sr = 44100
kemps = 32
nchnls = 2
0dbfs = 1

garvb init 0
gaW init 0
gaX init 0
gaY init 0

itmp ftgen 1, 0, 64, -2, 2, 40, -1, -1, -1, 123, \
          1, 13.000, 0.05, 0.85, 20000.0, 0.0, 0.50, 2, \
          1, 2.000, 0.05, 0.85, 20000.0, 0.0, 0.25, 2, \
          1, 16.000, 0.05, 0.85, 20000.0, 0.0, 0.35, 2, \
          1, 9.000, 0.05, 0.85, 20000.0, 0.0, 0.35, 2, \
          1, 12.000, 0.05, 0.85, 20000.0, 0.0, 0.35, 2, \
          1, 8.000, 0.05, 0.85, 20000.0, 0.0, 0.35, 2

itmp ftgen 2, 0, 262144, -2, 0
spat3dt 2, -0.2, 1, 0, 1, 1, 2, 0.005

itmp ftgen 3, 0, 262144, -52, 3, 2, 0, 4, 2, 1, 4, 2, 2, 4

instr 1

a1 vco2 1, 440, 10
kfrq port 100, 0.008, 20000
a1 butterlp a1, kfrq
a2 linseg 0, 0.003, 1, 0.01, 0.7, 0.005, 0, 1, 0
a1 = a1 * a2 * 2
denorm a1
vincr garvb, a1
aw, ax, ay, az spat3di a1, p4, p5, p6, 1, 1, 2
vincr gaW, aw
vincr gaX, ax
vincr gaY, ay

endin

instr 2

denorm garvb
; skip as many samples as possible without truncating the IR
arW, arX, arY ftconv garvb, 3, 2048, 2048, (65536 - 2048)
aW = gaW + arW
aX = gaX + arX
aY = gaY + arY
garvb = 0
gaW = 0
gaX = 0
gaY = 0

aWre, aWim hilbert aW
aXre, aXim hilbert aX
aYre, aYim hilbert aY
aWXr = 0.0928*aXre + 0.4699*aWre
aWXiYr = 0.2550*aXim - 0.1710*aWim + 0.3277*aYre
aL = aWXr + aWXiYr
aR = aWXr - aWXiYr

outs aL, aR

endin

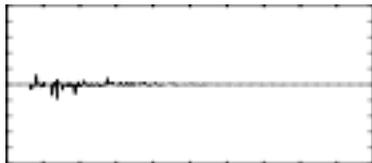
</CsInstruments>
<CsScore>

```

```
i 1 0 0.5 0.0 2.0 -0.8  
i 1 1 0.5 1.4 1.4 -0.6  
i 1 2 0.5 2.0 0.0 -0.4  
i 1 3 0.5 1.4 -1.4 -0.2  
i 1 4 0.5 0.0 -2.0 0.0  
i 1 5 0.5 -1.4 -1.4 0.2  
i 1 6 0.5 -2.0 0.0 0.4  
i 1 7 0.5 -1.4 1.4 0.6  
i 1 8 0.5 0.0 2.0 0.8  
i 2 0 10  
e
```

```
</CsScore>  
</CsoundSynthesizer>
```

This is the diagram of the waveform of the GEN52 routine, as used in the example:



```
f 3 0 262144 -52 3 2 0 4 2 1 4 2 2 4
```

## Credits

Author: Istvan Varga

# GENtanh

"tanh" — Generate a table with values on the tanh function.

## Description

Creates an ftable with values of the tanh function.

## Syntax

```
f # time size "tanh" start end rescale
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see *f statement*).

*start, end* -- first and last value to be stored; the GEN draws a curve that goes from start to end: tanh(start) ... tanh(end). The points stored are uniformly spaced between these to the table size

*rescale* -- if not zero the table is not rescaled

## Examples

Here is a simple example of the GENtanh routine. It uses the file *gentanh.csd* [examples/gentanh.csd].

### Example 1215. Example of the GENtanh routine.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac ;;realtime audio out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gentanh.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

aswp linseg 0.01, p3*.5, .49, p3*.5, 0.01 ;index sweep function
aindex poscil aswp, 110, 1 ;sound to waveshape
atable tablei aindex, p4, 1, .5 ;waveshape index
aenv linen 0.8, .01, p3, .02 ;amplitude envelope
asig = (atable*aenv)*p5 ;impose envelope and scale
asig dcblock2 asig ;get rid of DC
outs asig, asig
```

```

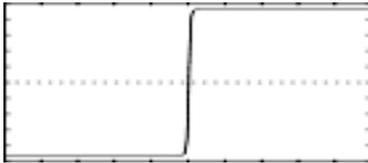
endin
</CsInstruments>
<CsScore>
f 1 0 8192 10 1 ;sine wave
f 2 0 8192 "tanh" -100 100 0 ;symmetrical transfer fuction
f 3 0 8192 "tanh" -10 10 0 ;symmetrical
f 4 0 8192 "tanh" 0 10 0 ;not symmetrical

i1 0 3 2 1
i1 + 3 3 1
i1 + 3 4 2

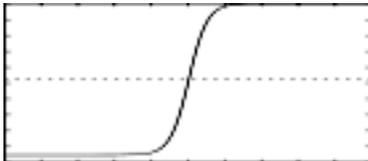
e
</CsScore>
</CsoundSynthesizer>

```

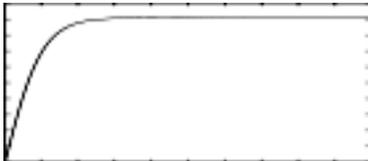
These are the diagrams of the waveforms of the GENtanh routines, as used in the example:



f 2 0 8192 "tanh" -100 100 0 - lots of distortion



f 3 0 8192 "tanh" -10 10 0 - less distortion than f2



f 4 0 8192 "tanh" -10 15 0

## See Also

*GENexp* and *GENsone*.

More information on this routine: <http://www.csoundjournal.com/issue11/distortionSynthesis.html>, written by Victor Lazzarini

## Credits

Written by John ffitich

# GENexp

"exp" — Generate a table with values on the exp function.

## Description

Creates an ftable with values of the exp function.

## Syntax

```
f # time size "exp" start end rescale
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see *f statement*).

*start, end* -- first and last value to be stored; the GEN draws a curve that goes from start to end: exp(start) ... exp(end). The points stored are uniformly spaced between these to the table size

*rescale* -- if not zero the table is not rescaled

## Examples

Here is a simple example of the GENexp routine. It uses the file *genexp.csd* [examples/genexp.csd].

### Example 1216. Example of the GENexp routine.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac ;;realtime audio out
;-iadc ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o genexp.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

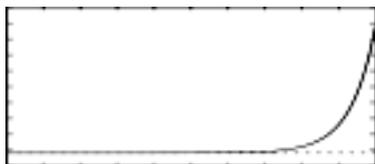
aswp linseg 0.01, p3*.5, .49, p3*.5, 0.01 ;index sweep function
aindex poscil aswp, 110, 1 ;sound to waveshape
atable tablei aindex, p4, 1, .5 ;waveshape index
aenv linen 0.8, .01, p3, .02 ;amplitude envelope
asig = (atable*aenv)*p5 ;impose envelope and scale
asig dcblock2 asig ;get rid of DC
outs asig, asig
```

```
endin
</CsInstruments>
<CsScore>
f 1 0 8192 10 1 ;sine wave
f 2 0 8192 "exp" 0 15 0
f 3 0 8192 "exp" 0 3 0

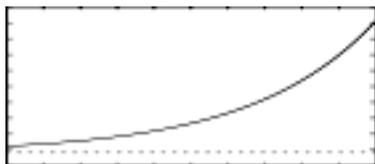
i1 0 3 2 2
i1 + 3 3 3

e
</CsScore>
</CsoundSynthesizer>
```

These are the diagrams of the waveforms of the GENexp routines, as used in the example:



f 2 0 8192 "exp" 0 15 0



f 3 0 8192 "exp" 0 3 0

## See Also

*GENexp* and *GENsone*.

More information on this routine: <http://www.csoundjournal.com/issue11/distortionSynthesis.html>, written by Victor Lazzarini

## Credits

Written by Victor Lazzarini

# GENsone

"sone" — Generate a table with values of the sone function.

## Description

Creates an ftable with values of the sone function for equal power.

## Syntax

```
f # time size "sone" start end equalpoint rescale
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see *f statement*).

*start, end* -- first and last value to be stored. The points stored are uniformly spaced between these to the table size.

*equalpoint* -- the point on the curve when the input and output values are equal.

*rescale* -- if not zero the table is not rescaled

The table is filled with the function  $x * \text{POWER}(x/\text{eqlp}, \text{FL}(33.0)/\text{FL}(78.0))$  for  $x$  between the start and end points. This is the Sone loudness curve.

## Examples

Here is an example of the GENsone routine. It uses the files *gensone.csd* [examples/gensone.csd].

### Example 1217. An example of the GENsone routine.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o gensone.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ; simple oscillator with loudness correction.

kcps = cpspch(p4)
kenv linseg 0, p3*0.25, 1, p3*0.75, 0 ;amplitude envelope
kamp tablei 16384 *kenv, 2
asig oscil kamp, kcps, 1
outs asig, asig
```

```

endin

instr 2 ;neutral oscillator to compare with

kcps = cpspch(p4)
kenv linseg 0, p3*0.25, 1, p3*0.75, 0 ;amplitude envelope
asig oscil kenv, kcps, 1
      outs asig, asig

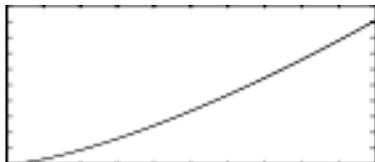
endin

</CsInstruments>
<CsScore>
f 1 0 16384 10 1 ;sine wave
f 2 0 16385 "sone" 0 32000 32000 0

s
f 0 1 ;1 second of silence before we start...
s
i 1 0 2 7.00
i 1 + . 7.01
i 1 + . 8.02
i 1 + . 8.03
s
i 2 0 2 7.00
i 2 + . 7.01
i 2 + . 8.02
i 2 + . 8.03
e
</CsScore>
</CsoundSynthesizer>

```

This is the diagram of the waveforms of the GENsone routine, as used in the example:



```
f 2 0 16385 "sone" 0 32000 32000 0
```

## See Also

More information on Sone: <http://en.wikipedia.org/wiki/Sone>

## Credits

Written by John ffitch

# GENquadbezier

"quadbezier" — Generate a table with values from a quadratic Bézier function.

## Description

This subroutine creates an ftable from segments constructed by paths traced by the quadratic Bézier function.

## Syntax

```
f # time size "quadbezier" y1 cx1 cy1 x2 y2 [cx2 cy2 x3 y3 ...]
```

## Initialization

*x2*, *x3*, etc. -- Locations in table at which to attain the following *y* value. Must be in increasing order. It is assumed that *x1* equals 0. If the last value is less than *size*, then the rest will be set to zero. Should not be negative.

*y1*, *y2*, *y3*, etc. -- Breakpoint values attained at the location specified by the preceding *x* value.

*cx1*, *cx2*, *cx3*, etc. -- Virtual *x* coordinates for the different control points. *cx[n]* can be equal to but not less than *x[n]* and equal to but not greater than *x[n+1]*. Should not be negative but can be zero.

*cy1*, *cy2*, *cy3*, etc. -- Virtual *y* coordinates for the different control points.

## Examples

Here is a simple example of the GENquadbezier routine. It uses the file *genquadbezier.csd* [examples/genquadbezier.csd].

### Example 1218. Example of the GENquadbezierroutine.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o quadbezier.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
```

```

instr 1
kndx phasor 1/p3
kenv tablei kndx, p4, 1
asig poscil kenv, 440, 1
    outs asig, asig

endin

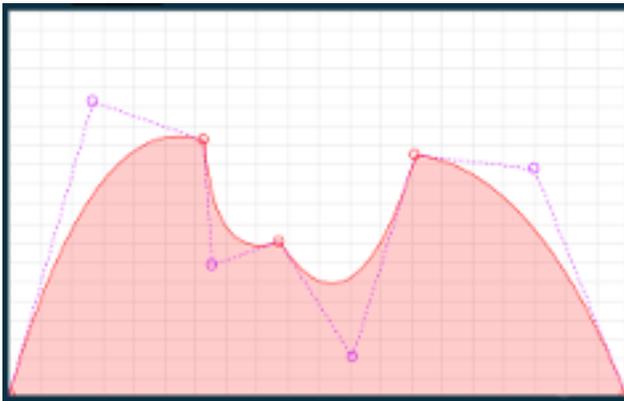
</CsInstruments>
<CsScore>
f 1 0 32768 10 1
f 2 0 1024 "quadbezier" 0 140 0.61 324 0.53 338 0.27 449 0.32 571 0.08 675 0.5 873 0.47 1024 0
f 3 0 1024 "quadbezier" 0 92 0.04 94 0.25 177 0.58 373 0.39 537 0.15 675 0.5 910 0.68 1024 0
f 4 0 1024 "quadbezier" 0 196 0.68 537 0.71 873 0.7 1024 0

i 1 0 4 2
i 1 4 4 3
i 1 8 4 4
</CsScore>

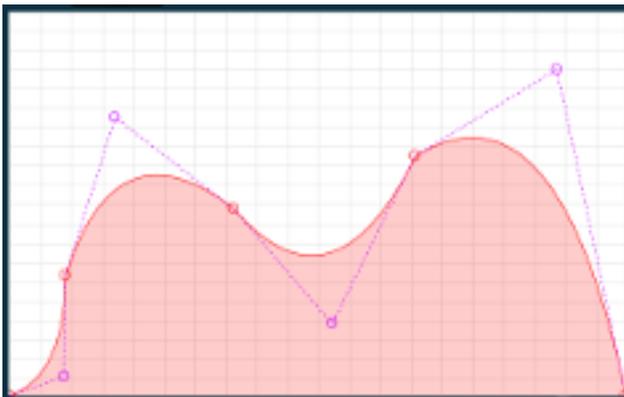
</CsoundSynthesizer>

```

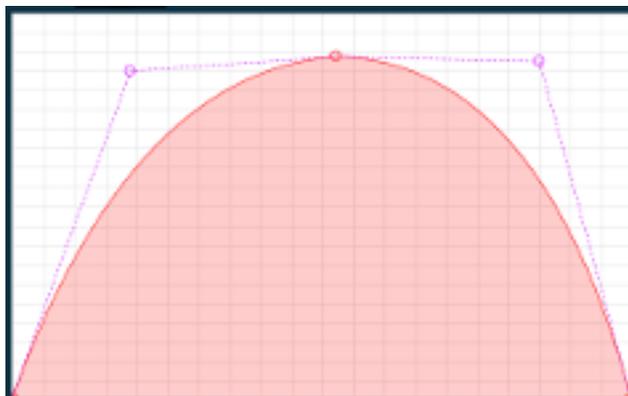
These are the diagrams of the envelopes drawn by the GENquadbezier routine, as used in the example above:



```
f 2 0 1024 "quadbezier" 0 140 0.61 324 0.53 338 0.27 449 0.32 571 0.08 675 0.5 873 0.47 1024 0
```



```
f 3 0 1024 "quadbezier" 0 92 0.04 94 0.25 177 0.58 373 0.39 537 0.15 675 0.5 910 0.68 1024 0
```



```
f 4 0 1024 "quadbezier" 0 196 0.68 537 0.71 873 0.7 1024 0
```

## See also

For references, check the comments in the *source code* [<https://github.com/csound/csound/blob/develop/Opcodes/quadbezier.c>].

## Credits

Written by Guillermo Senna

2016

# GENfarey

"farey" — Fills a table with the Farey Sequence  $F_n$  of the integer  $n$ .

## Description

A Farey Sequence  $F_n$  of order  $n$  is a list of fractions in their lowest terms between 0 and 1 and in ascending order. Their denominators do not exceed  $n$ . This means a fraction  $a/b$  belongs to  $F_n$  if  $0 < a \leq b \leq n$ . The numerator and denominator of each fraction are always coprime. 0 and 1 are included in  $F_n$  as the fractions  $0/1$  and  $1/1$ . For example  $F_5 = \{0/1, 1/5, 1/4, 1/3, 2/5, 1/2, 3/5, 2/3, 3/4, 4/5, 1/1\}$  Some properties of the Farey Sequence:

- If  $a/b$  and  $c/d$  are two successive terms of  $F_n$ , then  $bc - ad = 1$ .
- If  $a/b, c/d, e/f$  are three successive terms of  $F_n$ , then:  $c/d = (a+e) / (b+f)$ . In this case  $c/d$  is called the median fraction between  $a/b$  and  $e/f$ .
- If  $n > 1$ , then no two successive terms of  $F_n$  have the same denominator.

The length of any Farey Sequence  $F_n$  is determined by  $|F_n| = 1 + \text{SUM over } n (\text{phi}(m))$  where  $\text{phi}(m)$  is Euler's totient function, which gives the number of integers  $\leq m$  that are coprime to  $m$ .

Some values for the length of  $F_n$  given  $n$ :

$n$	$F_n$
1	2
2	3
3	5
4	7
5	11
6	13
7	19
8	23
9	29
10	33
11	43
12	47
13	59
14	65
15	73
16	81
17	97
18	103
19	121
20	129

## Syntax

```
f # time size "farey" fareynum mode
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see *f statement*).

*fareynum* -- the integer  $n$  for generating Farey Sequence  $F_n$

*mode* -- integer to trigger a specific output to be written into the table:

- 0 -- outputs floating point numbers representing the elements of  $F_n$ .
- 1 -- outputs delta values of successive elements of  $F_n$ , useful for generating note durations for example.
- 2 -- outputs only the denominators of the integer ratios, useful for indexing other tables or instruments for example.
- 3 -- same as mode 2 but with normalised output.
- 4 -- same as mode 0 but with 1 added to each number, useful for generating tables for tuning opcodes, for example *cps2pch*.

## Examples

```
f1 0 -23 "farey" 8 0
```

Generates generates Farey Sequence  $F_8$ . The table contains all 23 elements of  $F_8$  as floating point numbers.

```
f1 0 -18 "farey" 7 1
```

This generates Farey Sequence  $F_7$ . The table contains 18 delta values of  $F_7$ , i.e. the difference between  $r_{i+1} - r_i$ , where  $r$  is the  $i$ th element of  $F_n$ .

```
f1 0 -43 "farey" 11 2
```

This generates Farey Sequence  $F_{11}$ . The table contains the denominators of all 43 fractions in  $F_{11}$ .

```
f1 0 -43 "farey" 11 3
```

This generates Farey Sequence  $F_{11}$ . The table contains the denominators of all 43 fractions in  $F_{11}$ , each of those divided by 11, i.e. normalised.

```
f1 0 -18 "farey" 7 4
```

This generates Farey Sequence  $F_7$ . The table contains all fractions of  $F_7$ , same as mode 0, but this time '1' is added to each table element.

### Example 1219. A simple example of the GENfarey routine.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
```

```

<CsInstruments>

sr=44100
ksmps=10
nchnls=1

instr 4
  kndx init 0 ; read out elements of F_8 one by one and print to file
  if (kndx < 23) then
    kelem tab kndx, 1
    fprintks "farey8table.txt", "%.2f\\n", kelem
    kndx = kndx+1
  endif
endin
</CsInstruments>
<CsScore>
; initialise integer for Farey Sequence F_8
f1 0 -23 "farey" 8 0
; if mode=0 then the table stores all elements of the Farey Sequence
; as fractions in the range [0,1]
i4 0 1
e
</CsScore>
</CsoundSynthesizer>

```

Here is a complete example of the GENfarey routine. It uses the files *genfarey-2.csd* [examples/genfarey-2.csd].

### Example 1220. Another example of the GENfarey routine.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
-iadc      ;;uncomment -iadc for RT audio input as well
; For Non-realtime ouput leave only the line below:
; -o genfarey.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

; GENfarey creates table gidelta.
; The table contains the delta values of Farey Sequence 7 (p5=7).
; They are used as Inter Onset Intervals (IOIs) or event durations.
; If p6 is set to 1 for IOI output then the length of the table (p3=-18) is -(|F_7| - 1)
; Remember that a negative sign is for non-power-of-2 table lengths.
; The negative sign in front of the GEN number prevents post-normalisation of its values.

gidelta ftgen 0,0,-18,"farey",7,1

; Use GENfarey with p6 set to 2 to generate the denominators of fractions of F_7
; this is used in this example as factors to create a series of pitches:
gimult ftgen 0,0,-18,"farey",7,2

;----- loop and trigger instrument 901 using a Farey Sequence polyrhythm
instr 1
  kindx init 0
  kindx2 init 0
  ktrigger init 0

```

```

ktime_unit init p6
kstart init p4
kloop init p5
kinitndx init 0
kfn_times init gidelta
knote init 60
kbasenote init p8
ifundam init p7
ktrigger seqtime ktime_unit, kstart, kloop, kinitndx, kfn_times
  if (ktrigger > 0 ) then
    kpitch = cpspch(ifundam)
    kmult tab kindx2, gmult
    kpitch = kpitch * kmult
    knote = kbasenote + kmult
    event "i", 901, 0, .4, .10, kpitch, kpitch * .9, 0.4, 5, .75, .8, 1.0, .15, .0, .125, .12
    kindx = kindx + 1
    kindx = kindx % kloop
    kindx2 = kindx2 + 1
    kindx2 = kindx2 % kloop
  endif
endif

;----- basic 2 Operators FM algorithm -----
instr 901
inotedur = p3
imaxamp = p4 ;ampdb(p4)
icarrfreq = p5
imodfreq = p6
ilowndx = p7
indxdiff = p8-p7
knote = p27
aampenv linseg p9, p14*p3, p10, p15*p3, p11, p16*p3, p12, p17*p3, p13
adevenv linseg p18, p23*p3, p19, p24*p3, p20, p25*p3, p21, p26*p3, p22
amodosc oscili (ilowndx+indxdiff*adevenv)*imodfreq, imodfreq, 10
acarosc oscili imaxamp*aampenv, icarrfreq+amodosc, 10
outs acarosc, acarosc
endif
</CsInstruments>
<CsScore>
f10 0 4096 10 1 ;sine wave
; p4 kstart := index offset into the Farey Sequence
; p5 kloop := end index into Farey Seq.
; p6 timefac := time in seconds for one loop to complete
; p7 fundam := fundamental of the FM instrument
; p8 basenote:= root pitch of the midi voice output
; note that pitch structures of the midi file output are not equivalent to the
; ones used for the FM real-time synthesis.

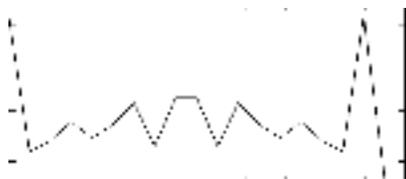
; start dur kstart kloop timefac fundam. basenote
i1 0.0 44 0 18 2 6.05 60
i1 4 30 0 18 3 7.05 72
i1 34 12 9 18 3 7.05 72
i1 10 12 0 18 1.5 8 84
i1 22 12 0 9 1.5 8 84
i1 15 16 0 18 1 5 48
i1 22 20 5 17 1.7 4 36

i1 46 20 3 11 2.5 7.04 71
i1 51 20 5 13 2.5 7.06 72

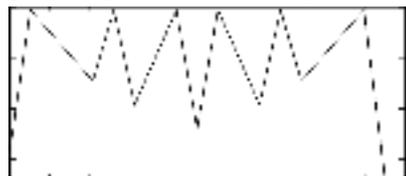
i1 73.5 1.5 11 18 1.5 5.05 48
i1 75 1 12 18 1 6.03 58
e
</CsScore>
</CsoundSynthesizer>

```

These are the diagrams of the waveforms of the GENfarey routines, as used in the example:



gidelta ftgen 100,0,-18,"farey",7,1 - delta values of Farey Sequence 7



gimult ftgen 101,0,-18,"farey",7,2 - generate the denominators of fractions of F\_7

## Credits

Author: Georg Boenn  
University of Glamorgan  
2010

New in Csound version 5.13

# GENwave

"wave" — Generates a compactly supported wavelet function.

## Description

Creates a compactly supported wavelet, scaling function or wavelet packet. The output function is obtained by deconvolution of corresponding mirror filter impulse response. This procedure is applied in an iterative fashion.

The filter bank used in classic discrete wavelet transform grows only towards low frequencies. Meanwhile wavelet packet transform introduces any possible direction of tree's growth. The sequence of mirror filters used in deconvolution is determined by binary form of seq value. "0" corresponds to low-pass filter and "1" to high-pass.

The number of iteration steps is determined by filter's length and size of function table. I.e. for filter length 8 and table size 256 there are  $\log_2(256/8) = 5$  iterations available.

## Syntax

```
f # time size "wave" fnsf seq rescale
```

## Initialization

*size* -- number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see *f statement*).

*fnsf* -- pre-existing table with scaling function coefficients.

*seq* -- non-negative integer number which corresponds to sequence of low-pass and high-pass mirror filters during deconvolution procedure.

*rescale* -- if not zero the table is not rescaled

## Examples

Here is an example of the GENwave routine. It uses the file *genwave.csd* [examples/genwave.csd].

### Example 1221. Example of the GENwave routine.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac          -iadc    -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o 0dbfs.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
nchnls = 1
0dbfs = 1
```

```

zakinit 3,1

instr 1 ; wavelet synth instrument
iamp = p4; scaling factor of wavelets
ifreq = p5; frequency of wavelets
itab = p6; selected wavelet function
inum = p7; number of wavelets to be created
a1 osciln p4, p5, p6, p7
out a1
endin

instr 2 ; wavelet analysis instrument
a1 soundin "fox.wav"
; Decomposition Structure:
; 1 LEVEL 2 LEVEL
; HP->ah1
; a1->| HP(up2)->ah2
; LP->a11->|
; LP(up2)->a12
;
ain = a1*.5; attenuate input signal
; since wavelet coefficients
; could reach big values
ah1 dconv ain,ftlen(8),8
al1 dconv ain,ftlen(7),7
ah2 dconv al1,ftlen(10),10
al2 dconv al1,ftlen(9),9

zaw ah1,0
zaw al1,1
zaw ah2,2
zaw al2,3

aout zar p4
out aout
zacl 0,3
endin

</CsInstruments>
<CsScore>

; First of all, we need several FIR filters which are capable
; to produce wavelet families.
; One can input filter coefficients manually using GEN02
; or read them from text file.
; Most of compact-supported wavelet coefficients can be obtained from
; Wavelet Browser by PyWavelets wavelets.pybytes.com
; You can select family and order of filter
; then copy desired coefficients into txt file.
; Notice that for correct interpretation of results you should use
; coeffs of Decomposition low-pass filter.

; Daubechies 2
f 1 0 4 -2 -0.1294095226 0.2241438680 0.8365163037 0.4829629131
; Symlet 10
f 2 0 0 -23 "sym10.txt"

; Now we want to produce some wavelet granules.
; They can be used in wavelet synthesis etc.
; Tables of large sizes should produce smoother wavelets.
; We take array of filter coefficients from ftable 1
; and deconvolve it until output length of 16384.
; The order of filters through the deconvolution process
; is given by 14 which is 1110 in binary.
; So the first filter is LP ('0') and others are HP ('1').
f 3 0 16384 "wave" 1 14 0

```

```
f 4 0 16384 "wave" 2 1 0
f 5 0 16384 "wave" 2 7 0
f 6 0 16384 "wave" 2 6 0
; The main purpose of using wavelets is wavelet transform.
; It is not that easy to perform a classic DWT in Csound since downsampling
; of audio signal is needed at each step of wavelet decomposition.
; Anyway, using GENwave it is possible to create a number of upsampled
; wavelets and perform a so-called undecimated wavelet transform
; aka stationary wavelet transform (and it is even better).
; So we need some upsampled childs of mother wavelet.
f 7 0 16 "wave" 1 0 -1 ;db2 scaling function for 1st iteration
f 8 0 16 "wave" 1 1 -1 ;db2 wavelet function for 1st iteration
f 9 0 32 "wave" 1 0 -1 ;db2 scaling function for 2nd iteration
f 10 0 32 "wave" 1 1 -1 ;db2 wavelet function for 2nd iteration

; Let's hear how some wavelets could sound..
; amp frq wave times
i 1 0 1 0.6 15 3 8
i 1 0.5 . 0.9 20 4 5
i 1 0.9 . 0.7 8 5 .
i 1 1.1 . 0.4 30 6 9

; Now try to decompose input file using wavelets
i 2 2 4 1; approximation 1st level
i 2 5 . 2; details 2nd level
</CsScore>
</CsoundSynthesizer>
```

## See Also

Ingrid Daubechies. Ten Lectures on Wavelets, SIAM 1992.

Pedro A.G. Batista. An Introduction to Sound Synthesis with Wavelet Packets, Csound Book CD Chapters, MIT 2000.

Victor M. Wickerhauser. Acoustic Signal Compression with Wavelet Packets, Yale 1992.

## Credits

Written by Gleb Rogozinsky

Saint-Petersburg University of Film and Television

Saint-Petersburg, Russia

April 2012

# GENpadsynth

"padsynth" — Generate a sample table using the padsynth algorithm.

## Description

Paul Octavian Nasca's "padsynth algorithm" adds bandwidth to each partial of a periodic waveform. This bandwidth is heard as color, movement, and additional richness of sound.

First, the waveform is defined by the user as a series of harmonic partials. Then, bandwidth is added by independently spreading each partial of the original waveform from a single frequency across neighboring frequencies, according to a "profile" function: a Gaussian curve, a square, or a rising and then falling exponential.

The partials of the original waveform may be considered to be samples in a discrete Fourier transform of the waveform. Normally there is not an exact one-to-one correspondence between the frequencies of the samples (frequency bins) of the discrete Fourier transform with the frequencies of the partials of the original waveform, because any frequency in the inverse of the discrete Fourier transform might be synthesized by interference between any number of bins. However, the padsynth algorithm uses a simple trick to create this correspondence. The discrete Fourier transform is simply made so large that the frequency of any partial of the original waveform will be very close to the frequency of the corresponding bin in the Fourier transform. Once this correspondence has been created, the bandwidth profile can be applied by centering it over the frequency bin of the original partial, scaling the profile by the bandwidth, and simply multiplying the original partial by each sample of the profile and adding the product to the corresponding bin of the Fourier transform.

As the frequencies of the partials increase, their bandwidth may optionally become wider or (less often) narrower.

Once each partial has been spread out in this way, the discrete Fourier transform may be given random phases, and is then simply inverted to synthesize the desired waveform, which may be used as the wavetable for a digital oscillator.

N.B.: The size of the function table does NOT necessarily reflect one periodic cycle of the waveform that it contains. The fundamental frequency must be used to generate the desired pitch from an oscillator using the function table, e.g.

```
oscillator_hz = desired_hz * (sr / padsynth_size / fundamental_hz)
```

## Syntax

```
f # score_time table_size "padsynth" fundamental_frequency
  partial_bandwidth partial_scale harmonic_stretch profile_shape profile_shape_parameter
  partial1_amplitude [partial2_amplitude ...]
```

## Initialization

*table\_size* -- Function table size. Should be large, e.g.  $2^{18} == 262144$ . Must be a power of 2 or power-of-2 plus 1 (see *f statement*).

*fundamental\_frequency* -- Fundamental frequency for the generated table.

*partial\_bandwidth* -- Bandwidth of each partial in cents.

*partial\_scale* -- Scaling factor for bandwidth of each partial (log of increase/decrease with partial frequency, 0 is no stretch or shrink).

*harmonic\_stretch* -- Harmonic stretch/shrink for all partials (1 is harmonic).

*profile\_shape* -- Number specifying the shape of the bandwidth profile: 1 = Gaussian, 2 = square, and 3 = exponential

*profile\_shape\_parameter* -- Parameter passed to the function generating the profile shape, e.g. exponent.

*partial1\_amplitude*, *partial2\_amplitude*, ... -- Amplitudes for each partial (may be zero).

## Examples

Here is an example of the GENpadsynth routine. It uses the files *padsynth\_gen.csd* [examples/padsynth\_gen.csd].

### Example 1222. An example of the GENpadsynth routine.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
sr=44100
ksmps=1
nchnls=2
0dbfs=2000

gispec_len init 2^18

instr 1
prints "Plain sine for frequency/amplitude/distortion comparison.\n"
gi_padsynth_1 ftgenonce 0, 0, gispec_len, 10, 1
iattack = 0.08
idecay = 0.1
isustain = 0.25
irelease = 0.2
aenv madsr iattack, idecay, isustain, irelease
ifreq cpsmidinn p4
iamp ampdb p5
ibasefreq = 440 ; can be lower or higher frequency; close to played frequency is said to be best
ibw_cents = 56.96943 ; width of the peaks, 100 is semitone
asig poscil iamp, ifreq, gi_padsynth_1
asig = aenv * asig
aleft, aright pan2 asig, 0.5
outs aleft, aright
endin

instr 2
prints "PadSynth with sine tone.\n"
ibasehz = 261.625565
;
p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11
gi_padsynth_1 ftgenonce 0, 0, gispec_len, "padsynth", ibasehz, p6, 0.0, 1, 1, 1.0, 1
iattack = 0.08
idecay = 0.1
isustain = 0.25
irelease = 0.2
aenv madsr iattack, idecay, isustain, irelease
ifreq cpsmidinn p4
iamp ampdb p5
asig poscil iamp, ifreq*(sr/gispec_len/ibasehz), gi_padsynth_1
asig = aenv * asig
aleft, aright pan2 asig, 0.5
outs aleft, aright
endin
```

```

instr 3
  prints "PadSynth with harmonics.\n"
  ibasehz = 261.625565
  ;
  gi_padsynth_1 ftgenonce 0, 0, gispec_len, "padsynth", ibasehz, p6, 1, 1, 1, 1, 0.7600046992, 0.61999
  iattack = 0.08
  idecay = 0.1
  isustain = 0.25
  irelease = 0.2
  aenv madsr iattack, idecay, isustain, irelease
  ifreq cpsmidinn p4
  iamp ampdb p5
  asig poscil iamp, ifreq*(sr/gispec_len/ibasehz), gi_padsynth_1
  asig = aenv * asig
  aleft, aright pan2 asig, 0.5
  outs aleft, aright
endin

```

```

instr 4
  prints "PadSynth with inharmonic partials.\n"
  ibasehz = 261.625565
  ;
  gi_padsynth_1 ftgenonce 0, 0, gispec_len, "padsynth", ibasehz, p6, 1, 2, 3, 1, 0.7600046992, 0.61999
  iattack = 0.08
  idecay = 0.1
  isustain = 0.25
  irelease = 0.2
  aenv madsr iattack, idecay, isustain, irelease
  ifreq cpsmidinn p4
  iamp ampdb p5
  asig poscil iamp, ifreq*(sr/gispec_len/ibasehz), gi_padsynth_1
  asig = aenv * asig
  aleft, aright pan2 asig, 0.5
  outs aleft, aright
endin

```

</CsInstruments>

<CsScore>

```

i1 0 2 60.00 60
i1 + 2 72.00 60
i1 + 2 84.00 60

i2 7 2 60.00 60 0.3
i2 + 2 72.00 60 0.3
i2 + 2 84.00 60 0.3
i2 + 2 60.00 60 25
i2 + 2 72.00 60 25
i2 + 2 84.00 60 25
i2 + 2 60.00 60 55
i2 + 2 72.00 60 55
i2 + 2 84.00 60 55

i3 26 2 60.00 60 0.3
i3 + 2 72.00 60 0.3
i3 + 2 84.00 60 0.3
i3 + 2 60.00 60 25
i3 + 2 72.00 60 25
i3 + 2 84.00 60 25
i3 + 2 60.00 60 55
i3 + 2 72.00 60 55
i3 + 2 84.00 60 55

```

```
i4 45 2 60.00 60 0.3
i4 + 2 72.00 60 0.3
i4 + 2 84.00 60 0.3
i4 + 2 60.00 60 25
i4 + 2 72.00 60 25
i4 + 2 84.00 60 25
i4 + 2 60.00 60 55
i4 + 2 72.00 60 55
i4 + 2 84.00 60 55

e
</CsScore>
</CsoundSynthesizer>
```

## See Also

More information on Padsynth: [www.paulnasca.com/algorithms-created-by-me](http://www.paulnasca.com/algorithms-created-by-me)

## Credits

Written by Michael Gogins

New in version 6.05

---

# Experimental Orchestra Opcodes and GEN Routines

## Experimental Orchestra Opcodes

The following opcodes require a nVidia GPU to be present. They are distributed with the hope that they may be useful, and to encourage the use of parallelism in audio processing. Some of the new opcodes perform the same process as conventional existing does but some are new.

# cudaanal

cudaanal — Generate an fsig from a mono audio source ain, using phase vocoder overlap-add analysis and GPU hardware. Experimental and only available as source code at the moment.

## Description

Generate an fsig from a mono audio source ain, using phase vocoder overlap-add analysis and GPU hardware.

## Syntax

```
fsig cudaanal ain, ifftsize, ioverlap, iwinsize, iwintype [, iformat] [, iinit]
```

## Initialization

*ifftsize* -- The FFT size in samples. Need not be a power of two (though these are especially efficient), but must be even. Odd numbers are rounded up internally. *ifftsize* determines the number of analysis bins in *fsig*, as  $ifftsize/2 + 1$ . For example, where  $ifftsize = 1024$ , *fsig* will contain 513 analysis bins, ordered linearly from the fundamental to Nyquist. The fundamental of analysis (which in principle gives the lowest resolvable frequency) is determined as  $sr/ifftsize$ . Thus, for the example just given and assuming  $sr = 44100$ , the fundamental of analysis is 43.07Hz. In practice, due to the phase-preserving nature of the phase vocoder, the frequency of any bin can deviate bilaterally, so that DC components are recorded. Given a strongly pitched signal, frequencies in adjacent bins can bunch very closely together, around partials in the source, and the lowest bins may even have negative frequencies.

As a rule, the only reason to use a non power-of-two value for *ifftsize* would be to match the known fundamental frequency of a strongly pitched source. Values with many small factors can be almost as efficient as power-of-two sizes; for example: 384, for a source pitched at around low A=110Hz.

*ioverlap* -- The distance in samples (“hop size”) between overlapping analysis frames. As a rule, this needs to be at least  $ifftsize/4$ , e.g. 256 for the example above. *ioverlap* determines the underlying analysis rate, as  $sr/ioverlap$ . *ioverlap* does not require to be a simple factor of *ifftsize*; for example a value of 160 would be legal. The choice of *ioverlap* may be dictated by the degree of pitch modification applied to the *fsig*, if any. As a rule of thumb, the more extreme the pitch shift, the higher the analysis rate needs to be, and hence the smaller the value for *ioverlap*. A higher analysis rate can also be advantageous with broadband transient sounds, such as drums (where a small analysis window gives less smearing, but more frequency-related errors).

Note that it is possible, and reasonable, to have distinct fsigs in an orchestra (even in the same instrument), running at different analysis rates. Interactions between such fsigs is currently unsupported, and the fsig assignment opcode does not allow copying between fsigs with different properties, even if the only difference is in *ioverlap*. However, this is not a closed issue, as it is possible in theory to achieve crude rate conversion (especially with regard to in-memory analysis files) in ways analogous to time-domain techniques.

*iwinsize* -- The size in samples of the analysis window filter (as set by *wintype*). This must be at least *ifftsize*, and can usefully be larger. Though other proportions are permitted, it is recommended that *iwinsize* always be an integral multiple of *ifftsize*, e.g. 2048 for the example above. Internally, the analysis window (Hamming, von Hann) is multiplied by a sinc function, so that amplitudes are zero at the boundaries between frames. The larger analysis window size has been found to be especially important for oscillator bank resynthesis (e.g. using *cuadsyn*), as it has the effect of increasing the frequency resolution of the analysis, and hence the accuracy of the resynthesis. As noted above, *iwinsize* determines the overall latency

of the analysis/resynthesis system. In many cases, and especially in the absence of pitch modifications, it will be found that setting *iwin*size=*ifft*size works very well, and offers the lowest latency.

*iwin*type -- The shape of the analysis window. Currently only two choices are implemented:

- 0 = Hamming window
- 1 = von Hann window

Both are also supported by the PVOC-EX file format. The window type is stored as an internal attribute of the fsig, together with the other parameters (see *pvsinfo*). Other types may be implemented later on (e.g. the Kaiser window, also supported by PVOC-EX), though an obvious alternative is to enable windows to be defined via a function table. The main issue here is the constraint of f-tables to power-of-two sizes, so this method does not offer a complete solution. Most users will find the Hamming window meets all normal needs, and can be regarded as the default choice.

*iform*at -- (optional) The analysis format. Currently only one format is implemented by this opcode:

- 0 = amplitude + frequency

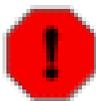
This is the classic phase vocoder format; easy to process, and a natural format for oscillator-bank resynthesis. It would be very easy (tempting, one might say) to treat an fsig frame not purely as a phase vocoder frame but as a generic additive synthesis frame. It is indeed possible to use an fsig this way, but it is important to bear in mind that the two are not, strictly speaking, directly equivalent.

Other important formats (supported by PVOC-EX) are:

- 1 = amplitude + phase
- 2 = complex (real + imaginary)

*iform*at is provided in case it proves useful later to add support for these other formats. Formats 0 and 1 are very closely related (as the phase is “wrapped” in both cases - it is a trivial matter to convert from one to the other), but the complex format might warrant a second explicit signal type (a “csig”) specifically for convolution-based processes, and other processes where the full complement of arithmetic operators may be useful.

*iinit* -- (optional) Skip reinitialization. This is not currently implemented for any of these opcodes, and it remains to be seen if it is even practical.



## Warning

It is unsafe to use the same f-variable for both input and output of cuda opcodes. Using the same one might lead to undefined behavior on some opcodes. Use a different one on the left and right sides of the opcode.

## Examples

Here is an example of the *codanal* opcode. It uses the file *codanal.csd* [examples/codanal.csd].

### Example 1223. Example of the *codanal* opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o cudanal.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;cudanal has no influence when there is no transformation of original sound

ifftsize = p4
ioverlap = ifftsize / 4
iwinshape = 1 ;von-Hann window
Sfile = "fox.wav"
ain soundin Sfile
fftin cudanal ain, ifftsize, ioverlap, iwinshape ;fft-analysis of the audio-signal
fftblur pvscale fftin, p5 ;scale
aout cudasyntn fftblur ;resynthesis
outs aout, aout

endin

</CsInstruments>
<CsScore>
s
i 1 0 3 512 1 ;original sound - ifftsize of pvsanal does not have any influence
i 1 3 3 1024 1 ;even with different
i 1 6 3 2048 1 ;settings

s
i 1 0 3 512 1.5 ;but transformation - here a fifth higher
i 1 3 3 1024 1.5 ;but with different settings
i 1 6 3 2048 1.5 ;for ifftsize of pvsanal

e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
2013

New in version 6.02

# cudasynth

cudasynth — Synthesis by additive synthesis and inverse FFT. Experimental and only available as source code at the moment.

## Description

Synthesis by additive synthesis and inverse FFT.

## Syntax

```
asig cudasynth kamp, kfreq, itab, iftab, iatab[, inum]
```

```
asig cudasynth fsig, kamp, kfreq[, inum]
```

```
asig cudasynth fsig
```

## Initialization

*itab* --

*iftab* --

*iatab* --

*inum* -- (optional)

## Performance

*kamp* --

*kfreq* --

*fsig* --

## Examples

Here is an example of the cudasynth opcode. It uses the file *cudasynth.csd* [examples/cudasynth.csd].

### Example 1224. Example of the cudasynth opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>  
<CsOptions>  
; Select audio/midi flags here according to platform  
-odac      ;;realtime audio out  
;-iadc     ;;uncomment -iadc if realtime audio input is needed too  
; For Non-realtime ouput leave only the line below:  
; -o cudsyth.wav -W ;; for file output any platform  
</CsOptions>  
<CsInstruments>
```

```
sr = 44100
ksmps = 64
nchnls = 1
0dbfs = 1

i1 ftgen 1,0,512,7,1,512,0.001
i2 ftgen 2,0,512,-7,1,512,512
i3 ftgen 3,0,16384,10,1
schedule 1,0,10

instr 1
a1 cudasynth 0.001, 100,0, 2, 1,128
  out a1

endin
</CsInstruments>
<CsScore>
e 10
</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Victor Lazzarini  
2013

New in version 6.02

# cuasliding

cuasliding — Perform sliding phase vocoder algorithm with simplified transformational FM using GPU hardware. Experimental and only available as source code at the moment.

## Description

Perform sliding phase vocoder algorithm with simplified transformational FM using GPU hardware.

## Syntax

```
asig cuasliding ain, amod, iwinsize
```

## Initialization

*iwinsize* -- The DFT size in samples. Need not be a power of two. *ifftsize* determines the number of analysis bins used.

## Performance

*ain* -- input signal for transformation.

*amod* -- signal that modulates the analysed signal.

## Examples

Here is an example of the cuasliding opcode. It uses the file *cuasliding.csd* [examples/cuasliding.csd].

### Example 1225. Example of the cuasliding opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac      ;;realtime audio out
;-iadc     ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o cuaslidig.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 64
0dbfs = 1
nchnls = 2

instr 1
asig = diskin:a("flutec3.wav",1,0,1)
amod = oscil:a(1,3)
asig2 = cuasliding(asig,amod)
asig = linenr(asig2,0.005,0.01,0.01)
```

```
    out(asig)
  endin

  </CsInstruments>
  <CsScore>
  i1 0 60

  </CsScore>
  </CsoundSynthesizer>
```

## Credits

Author: Russell Bradford  
March 2014

New in version 6.03

---

# Deprecated Orchestra Opcodes and GEN Routines

## Deprecated Orchestra Opcodes

The following opcodes are deprecated. They are still distributed with Csound for backward compatibility. The new opcodes which are to be used in place of the deprecated ones are indicated within each of the following entries.

# abetarand

abetarand — Deprecated.

## Description

Deprecated as of version 3.49. Use the *betarand* opcode instead.

## abexprnd

abexprnd — Deprecated.

### Description

Deprecated as of version 3.49. Use the *bexprnd* opcode instead.

# acauchy

acauchy — Deprecated.

## Description

Deprecated as of version 3.49. Use the *cauchy* opcode instead.

# **aexprand**

aexprand — Deprecated.

## **Description**

Deprecated as of version 3.49. Use the *exprand* opcode instead.

## agauss

agauss — Deprecated.

### Description

Deprecated as of version 3.49. Use the *gauss* opcode instead.

# agogobel

agogobel — Deprecated.

## Description

New in version 3.47

Deprecated as of version 3.52. Use the *gogobel* opcode instead.

# alinrand

alinrand — Deprecated.

## Description

Deprecated as of version 3.49. Use the *linrand* opcode instead.

## **apcauchy**

apcauchy — Deprecated.

### **Description**

Deprecated as of version 3.49. Use the *pcauchy* opcode instead.

# apoisson

apoisson — Deprecated.

## Description

Deprecated as of version 3.49. Use the *poisson* opcode instead.

## **apow**

apow — Deprecated.

### **Description**

Deprecated as of version 3.48. Use the *pow* opcode instead.

# array

array — Deprecated.

## Description

Generate a vector (one-dimensional k-rate array) with initial values.

## Syntax

```
karray[] array ival1, ival2,....ivaln
```

## Initialization

*ival1,...ivaln* -- values to place in the vector.

## Examples

Here is an example of the array opcode. It uses the file *array.csd* [examples/array.csd].

### Example 1226. Example of the array opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-n
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

  kS[] = array(1,7,5)
  printk2 kS[0]
  printk2 kS[1]
  printk2 kS[2]
endin

</CsInstruments>
<CsScore>
i 1 0 0
e
</CsScore>
</CsoundSynthesizer>
```

## See Also

Vectorial opcodes fillarray

## Credits

Author: John ffitch  
Codemist Ltd  
2013

New in version 6.00

# atrirand

atrirand — Deprecated.

## Description

Deprecated as of version 3.49. Use the *trirand* opcode instead.

# aunirand

aunirand — Deprecated.

## Description

Deprecated as of version 3.49. Use the *unirand* opcode instead.

# aweibull

aweibull — Deprecated.

## Description

Deprecated as of version 3.49. Use the *weibull* opcode instead.

# bformdec

bformdec — Deprecated. Decodes an ambisonic B format signal.

## Description

Decodes an ambisonic B format signal into loudspeaker specific signals. Note that this opcode is deprecated as it is inaccurate, and is replaced by the much better opcode *bformdec1* which replicates all the important features.

## Syntax

```
ao1, ao2 bformdec isetup, aw, ax, ay, az [, ar, as, at, au, av \  
[, abk, al, am, an, ao, ap, aq]]  
  
ao1, ao2, ao3, ao4 bformdec isetup, aw, ax, ay, az [, ar, as, at, \  
au, av [, abk, al, am, an, ao, ap, aq]]  
  
ao1, ao2, ao3, ao4, ao5 bformdec isetup, aw, ax, ay, az [, ar, as, \  
at, au, av [, abk, al, am, an, ao, ap, aq]]  
  
ao1, ao2, ao3, ao4, ao5, ao6, ao7, ao8 bformdec isetup, aw, ax, ay, az \  
[, ar, as, at, au, av [, abk, al, am, an, ao, ap, aq]]]
```

## Initialization

*isetup* — loudspeaker setup. There are five supported setups: 1 denotes stereo setup. There must be two output cells with loudspeaker positions assumed to be (330/0, 30/0).

2 denotes quad setup. There must be four output cells. Loudspeaker positions assumed to be (45°/0), (135°/0), (225/0), (315/0).

3 is a 5.1 surround setup. There must be five output cells. LFE channel is not supported. Loudspeaker positions assumed to be (330/0), (30/0), (0/0), (250/0), (110/0).

4 denotes eight loudspeaker circle setup. There must be eight output cells. Loudspeaker positions assumed to be (22.5/0), (67.5/0), (112.5/0), (157.5/0), (202.5/0), (247.5/0), (292.5/0), (337.5/0).

5 means an eight loudspeaker cubic setup. There must be eight output cells. Loudspeaker positions assumed to be (45/0), (45/30), (135/0), (135/30), (225/0), (225/30), (315/0), (315/30).

## Performance

*aw, ax, ay, ...* -- input signal in the B format.

*ao1 .. ao8* — loudspeaker specific output signals.

## Example

Here is an example of the bformdec opcode. It uses the file *bformenc.csd* [examples/bformenc.csd].

### Example 1227. Example of the bformdec opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
;-odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime output leave only the line below:
-o bformenc.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>
sr = 44100
kr = 4410
ksmps = 10
nchnls = 8

; bformenc is deprecated, please use bformenc1

instr 1
; generate pink noise
anoise pinkish 1000

; two full turns
kalpha line 0, p3, 720
kbeta = 0

; fade ambisonic order from 2nd to 0th during second turn
kord0 = 1
kord1 linseg 1, p3 / 2, 1, p3 / 2, 0
kord2 linseg 1, p3 / 2, 1, p3 / 2, 0

; generate B format
aw, ax, ay, az, ar, as, at, au, av bformenc anoise, kalpha, kbeta, kord0, kord1, kord2

; decode B format for 8 channel circle loudspeaker setup
a1, a2, a3, a4, a5, a6, a7, a8 bformdec 4, aw, ax, ay, az, ar, as, at, au, av

; write audio out
outo a1, a2, a3, a4, a5, a6, a7, a8
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for 20 seconds.
i 1 0 20
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Samuel Groner  
2005

New in version 5.07. Deprecated in 5.09

# bformenc

bformenc — Deprecated. Codes a signal into the ambisonic B format.

## Description

Codes a signal into the ambisonic B format. Note that this opcode is deprecated as it is inaccurate, and is replaced by the much better opcode *bformenc1* which replicates all the important features; also note that the gain arguments are not available in *bformenc1*.

## Syntax

```
aw, ax, ay, az bformenc asig, kalpha, kbeta, kord0, kord1
aw, ax, ay, az, ar, as, at, au, av bformenc asig, kalpha, kbeta, \
    kord0, kord1 , kord2
aw, ax, ay, az, ar, as, at, au, av, ak, al, am, an, ao, ap, aq bformenc \
    asig, kalpha, kbeta, kord0, kord1, kord2, kord3
```

## Performance

*aw, ax, ay, ...* -- output cells of the B format.

*asig* -- input signal.

*kalpha* -- azimuth angle in degrees (clockwise).

*kbeta* -- altitude angle in degrees.

*kord0* -- linear gain of the zero order B format.

*kord1* -- linear gain of the first order B format.

*kord2* -- linear gain of the second order B format.

*kord3* -- linear gain of the third order B format.

## Example

Here is an example of the bformenc opcode. It uses the file *bformenc.csd* [examples/bformenc.csd].

### Example 1228. Example of the bformenc opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
;-odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
-o bformenc.wav -W ;; for file output any platform
```

```
</CsOptions>
<CsInstruments>
sr = 44100
kr = 4410
ksmps = 10
nchnls = 8

; bformenc is deprecated, please use bformenc1

instr 1
; generate pink noise
anoise pinkish 1000

; two full turns
kalpha line 0, p3, 720
kbeta = 0

; fade ambisonic order from 2nd to 0th during second turn
kord0 = 1
kord1 linseg 1, p3 / 2, 1, p3 / 2, 0
kord2 linseg 1, p3 / 2, 1, p3 / 2, 0

; generate B format
aw, ax, ay, az, ar, as, at, au, av bformenc anoise, kalpha, kbeta, kord0, kord1, kord2

; decode B format for 8 channel circle loudspeaker setup
a1, a2, a3, a4, a5, a6, a7, a8 bformdec 4, aw, ax, ay, az, ar, as, at, au, av

; write audio out
outo a1, a2, a3, a4, a5, a6, a7, a8
endin

</CsInstruments>
<CsScore>

; Play Instrument #1 for 20 seconds.
i 1 0 20
e

</CsScore>
</CsSoundSynthesizer>
```

## Credits

Author: Samuel Groner  
2005

New in version 5.07. Deprecated in 5.09.

# clock

clock — Deprecated.

## Description

Deprecated. Use the *rtclock* opcode instead.

# hrtfer

hrtfer — Creates 3D audio for two speakers.

## Description

Output is binaural (headphone) 3D audio.



### Note

This opcode is deprecated; use `hrtfstat` or similar opcode instead

## Syntax

```
aleft, aright hrtfer asig, kaz, kelev, "HRTFcompact"
```

## Initialization

*kAz* -- azimuth value in degrees. Positive values represent position on the right, negative values are positions on the left.

*kElev* -- elevation value in degrees. Positive values represent position above horizontal, negative values are positions under horizontal.

At present, the only file which can be used with *hrtfer* is *HRTFcompact* [examples/HRTFcompact]. It must be passed to the opcode as the last argument within quotes as shown above.

HRTFcompact may also be obtained via anonymous ftp from: `ftp://ftp.cs.bath.ac.uk/pub/dream/utilities/Analysis/HRTFcompact`

## Performance

These unit generators place a mono input signal in a virtual 3D space around the listener by convolving the input with the appropriate HRTF data specified by the opcode's azimuth and elevation values. *hrtfer* allows these values to be k-values, allowing for dynamic spatialization. *hrtfer* can only place the input at the requested position because the HRTF is loaded in at i-time (remember that currently, CSound has a limit of 20 files it can hold in memory, otherwise it causes a segmentation fault). The output will need to be scaled either by using *balance* or by multiplying the output by some scaling constant.



### Note

The sampling rate of the orchestra must be 44.1kHz. This is because 44.1kHz is the sampling rate at which the HRTFs were measured. In order to be used at a different rate, the HRTFs would need to be re-sampled at the desired rate.

## Examples

Here is an example of the *hrtfer* opcode. It uses the file *hrtfer.csd* [examples/hrtfer.csd], *HRTFcompact* [examples/HRTFcompact], and *beats.wav* [examples/beats.wav].

## Example 1229. Example of the hrtfer opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
; Audio out  Audio in  No messages
-odac      -iadc      -d      ;;RT audio I/O
; For Non-realtime ouput leave only the line below:
; -o hrtfer.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

instr 1
kaz      linseg 0, p3, -360 ; move the sound in circle
kel      linseg -40, p3, 45 ; around the listener, changing
; elevation as its turning

asrc      soundin "beats.wav"
aleft,right hrtfer asrc, kaz, kel, "HRTFcompact"
aleftscale = aleft * 200
arightscale = aright * 200

outs      aleftscale, arightscale
endin

</CsInstruments>
<CsScore>

i 1 0 2
e

</CsScore>
</CsoundSynthesizer>
```

## See Also

*hrtfmove, hrtfmove2, hrtfstat.*

## Credits

Authors: Eli Breder and David MacIntyre  
Montreal  
1996

Fixed the example thanks to a message from Istvan Varga.

# ibetarand

ibetarand — Deprecated.

## Description

Deprecated as of version 3.49. Use the *betarand* opcode instead.

## **ibexprnd**

ibexprnd — Deprecated.

### **Description**

Deprecated as of version 3.49. Use the *bexprnd* opcode instead.

# icauchy

icauchy — Deprecated.

## Description

Deprecated as of version 3.49. Use the *cauchy* opcode instead.

## ictrl14

ictrl14 — Deprecated.

### Description

Deprecated as of version 3.52. Use the *ctrl14* opcode instead.

## ictrl21

ictrl21 — Deprecated.

### Description

Deprecated as of version 3.52. Use the *ctrl21* opcode instead.

## ictrl7

ictrl7 — Deprecated.

### Description

Deprecated as of version 3.52. Use the *ctrl7* opcode instead.

# **iexprand**

iexprand — Deprecated.

## **Description**

Deprecated as of version 3.49. Use the *exprand* opcode instead.

# igauss

igauss — Deprecated.

## Description

Deprecated as of version 3.49. Use the *gauss* opcode instead.

# ilinrand

ilinrand — Deprecated.

## Description

Deprecated as of version 3.49. Use the *linrand* opcode instead.

## **imidic14**

imidic14 — Deprecated.

### **Description**

Deprecated as of version 3.52. Use the *midic14* opcode instead.

## imidic21

imidic21 — Deprecated.

### Description

Deprecated as of version 3.52. Use the *midic21* opcode instead.

## **imidic7**

*imidic7* — Deprecated.

### **Description**

Deprecated as of version 3.52. Use the *midic7* opcode instead.

# instimek

instimek — Deprecated.

## Description

Deprecated as of version 3.62. Use the *timeinstk* opcode instead.

## Credits

David M. Boothe originally pointed out this deprecated name.

# instimes

instimes — Deprecated.

## Description

Deprecated as of version 3.62. Use the *timeinsts* opcode instead.

## Credits

David M. Boothe originally pointed out this deprecated name.

## **ioff**

ioff — Deprecated.

### **Description**

Deprecated as of version 3.52. Use the *noteoff* opcode instead.

# ion

ion — Deprecated.

## Description

Deprecated as of version 3.52. Use the *noteon* opcode instead.

# iondur2

iondur2 — Deprecated.

## Description

Deprecated as of version 3.52. Use the *noteondur2* opcode instead.

# iondur

iondur — Deprecated.

## Description

Deprecated as of version 3.52. Use the *noteondur* opcode instead.

# ioutat

ioutat — Deprecated.

## Description

Deprecated as of version 3.52. Use the *outiat* opcode instead.

# ioutc14

ioutc14 — Deprecated.

## Description

Deprecated as of version 3.52. Use the *outc14* opcode instead.

# ioutc

ioutc — Deprecated.

## Description

Deprecated as of version 3.52. Use the *outic* opcode instead.

# ioutput

ioutput — Deprecated.

## Description

Deprecated as of version 3.52. Use the *outipat* opcode instead.

# ioutpb

ioutpb — Deprecated.

## Description

Deprecated as of version 3.52. Use the *outpb* opcode instead.

# ioutpc

ioutpc — Deprecated.

## Description

Deprecated as of version 3.52. Use the *outipc* opcode instead.

## ipcauchy

ipcauchy — Deprecated.

### Description

Deprecated as of version 3.49. Use the *pcauchy* opcode instead.

# ipoisson

ipoisson — Deprecated.

## Description

Deprecated as of version 3.49. Use the *poisson* opcode instead.

# ipow

ipow — Deprecated.

## Description

Deprecated as of version 3.48. Use the *pow* opcode instead.

## is16b14

is16b14 — Deprecated.

### Description

Deprecated as of version 3.52. Use the *s16b14* opcode instead.

# is32b14

is32b14 — Deprecated.

## Description

Deprecated as of version 3.52. Use the *s32b14* opcode instead.

## islider16

islider16 — Deprecated.

### Description

Deprecated as of version 3.52. Use the *slider16* opcode instead.

## islider32

islider32 — Deprecated.

### Description

Deprecated as of version 3.52. Use the *slider32* opcode instead.

## islider64

islider64 — Deprecated.

### Description

Deprecated as of version 3.52. Use the *slider64* opcode instead.

## islider8

islider8 — Deprecated.

### Description

Deprecated as of version 3.52. Use the *slider8* opcode instead.

# itablecopy

itablecopy — Deprecated.

## Description

Deprecated as of version 3.52. Use the *tablecopy* opcode instead.

# itablegpw

itablegpw — Deprecated.

## Description

Deprecated as of version 3.52. Use the *tableigpw* opcode instead.

## itablemix

itablemix — Deprecated.

### Description

Deprecated as of version 3.52. Use the *tablemix* opcode instead.

## itablew

itablew — Deprecated.

### Description

Deprecated as of version 3.52. Use the *tableiw* opcode instead.

# itrirand

itrirand — Deprecated.

## Description

Deprecated as of version 3.49. Use the *trirand* opcode instead.

# iunirand

iunirand — Deprecated.

## Description

Deprecated as of version 3.49. Use the *unirand* opcode instead.

# **iweibull**

iweibull — Deprecated.

## **Description**

Deprecated as of version 3.49. Use the *weibull* opcode instead.

# kbetarand

kbetarand — Deprecated.

## Description

Deprecated as of version 3.49. Use the *betarand* opcode instead.

## **kbexprnd**

kbexprnd — Deprecated.

### **Description**

Deprecated as of version 3.49. Use the *bexprnd* opcode instead.

# kcauchy

kcauchy — Deprecated.

## Description

Deprecated as of version 3.49. Use the *cauchy* opcode instead.

## **kdump2**

kdump2 — Deprecated.

### **Description**

Deprecated as of version 3.49. Use the *dumpk2* opcode instead.

## **kdump3**

kdump3 — Deprecated.

### **Description**

Deprecated as of version 3.49. Use the *dumpk3* opcode instead.

# kdump4

kdump4 — Deprecated.

## Description

Deprecated as of version 3.49. Use the *dumpk4* opcode instead.

# **kdump**

kdump — Deprecated.

## **Description**

Deprecated as of version 3.49. Use the *dumpk* opcode instead.

# kexprand

kexprand — Deprecated.

## Description

Deprecated as of version 3.49. Use the *exprand* opcode instead.

## kfilter2

kfilter2 — Deprecated.

### Description

Deprecated as of version 3.49. Use the *filter2* opcode instead.

### Credits

Author: Michael A. Casey  
M.I.T.  
Cambridge, Mass.  
1997

New in version 3.47

# kgauss

kgauss — Deprecated.

## Description

Deprecated as of version 3.49. Use the *gauss* opcode instead.

# klinrand

klinrand — Deprecated.

## Description

Deprecated as of version 3.49. Use the *linrand* opcode instead.

# kon

kon — Deprecated.

## Description

Deprecated as of version 3.49. Use the *midion* opcode instead.

# koutat

koutat — Deprecated.

## Description

Deprecated as of version 3.52. Use the *outkat* opcode instead.

# koutc14

koutc14 — Deprecated.

## Description

Deprecated as of version 3.52. Use the *outkc14* opcode instead.

# koutc

koutc — Deprecated.

## Description

Deprecated as of version 3.52. Use the *outkc* opcode instead.

# koutpat

koutpat — Deprecated.

## Description

Deprecated as of version 3.52. Use the *outpat* opcode instead.

# koutpb

koutpb — Deprecated.

## Description

Deprecated as of version 3.52. Use the *outkpb* opcode instead.

# koutpc

koutpc — Deprecated.

## Description

Deprecated as of version 3.52. Use the *outkpc* opcode instead.

# kpcauchy

kpcauchy — Deprecated.

## Description

Deprecated as of version 3.49. Use the *pcauchy* opcode instead.

# kpoisson

kpoisson — Deprecated.

## Description

Deprecated as of version 3.49. Use the *poisson* opcode instead.

# kpow

kpow — Deprecated.

## Description

Deprecated as of version 3.48. Use the *pow* opcode instead.

# kread2

kread2 — Deprecated.

## Description

Deprecated as of version 3.52. Use the *readk2* opcode instead.

## kread3

kread3 — Deprecated.

### Description

Deprecated as of version 3.52. Use the *readk3* opcode instead.

## kread4

kread4 — Deprecated.

### Description

Deprecated as of version 3.52. Use the *readk4* opcode instead.

# kread

kread — Deprecated.

## Description

Deprecated as of version 3.52. Use the *readk* opcode instead.

# **htableseg**

htableseg — Deprecated.

## **Description**

Deprecated. Use the *htableseg* opcode instead.

## **Syntax**

```
htableseg ifn1, idur1, ifn2 [, idur2] [, ifn3] [...]
```

# ktrirand

ktrirand — Deprecated.

## Description

Deprecated as of version 3.49. Use the *trirand* opcode instead.

# kunirand

kunirand — Deprecated.

## Description

Deprecated as of version 3.49. Use the *unirand* opcode instead.

# kweibull

kweibull — Deprecated.

## Description

Deprecated as of version 3.49. Use the *weibull* opcode instead.

# sndload

sndload — Loads a sound file into memory for use by *loscilx*

## Description

*sndload* loads a sound file into memory for use by *loscilx*.



### Note

This opcode loads the file into memory but it is not available for use afterwards. In effect it just wastes memory.

## Syntax

```
sndload Sfname[, ifmt[, ichns[, isr[, ibas[, iamp[, istr \
[, ilpmod[, ilps[, ilpe]]]]]]]]]]
```

## Initialization

*Sfname* - file name as a string constant or variable, string p-field, or a number that is used either as an index to strings set with *strset*, or, if that is not available, a file name in the format *soundin.n* is used. If the file name does not include a full path, the file is searched in the current directory first, then those specified by *SSDIR* (if defined), and finally *SFDIR*. If the same file was already loaded previously, it will not be read again, but the parameters *ibas*, *iamp*, *istr*, *ilpmod*, *ilps*, and *ilpe* are still updated.

*ifmt* (optional, defaults to zero) - default sample format for raw (headerless) sound files; if the file has a header, this is ignored. Can be one of the following:

- 1: do not allow headerless files (fail with an init error)
- 0: use the same format as the one specified on the command line
- 1: 8 bit signed integers
- 2: a-law
- 3: u-law
- 4: 16 bit signed integers
- 5: 32 bit signed integers
- 6: 32 bit floats
- 7: 8 bit unsigned integers
- 8: 24 bit signed integers
- 9: 64 bit floats

*ichns* (optional, defaults to zero) - default number of channels for raw (headerless) sound files; if the file has a header, this is ignored. Zero or negative values are interpreted as 1 channel.

*isr* (optional, defaults to zero) - default sample rate for raw (headerless) sound files; if the file has a header, this is ignored. Zero or negative values are interpreted as the orchestra sample rate (*sr*).

*ibas* (optional, defaults to zero) - base frequency in Hz. If positive, overrides the value specified in the sound file header; otherwise, the value from the header is used if present, and 1.0 if the file does not include such information.

*iamp* (optional, defaults to zero) - amplitude scale. If non-zero, overrides the value specified in the sound file header (note: negative values are allowed, and will invert the sound output); otherwise, the value from the header is used if present, and 1.0 if the file does not include such information.

*istrt* (optional, defaults to -1) - starting position in sample frames, can be fractional. If non-negative, overrides the value specified in the sound file header; otherwise, the value from the header is used if present, and 0 if the file does not include such information. Note: even if this parameter is specified, the whole file is still read into memory.

*ilpmod* (optional, defaults to -1) - loop mode, can be one of the following:

any negative value: use the loop information specified in the sound file header, ignoring *ilps* and *ilpe*

0: no looping (*ilps* and *ilpe* are ignored)

1: forward looping (wrap around loop end if it is crossed in forward direction, and wrap around loop start if it is crossed in backward direction)

2: backward looping (change direction at loop end if it is crossed in forward direction, and wrap around loop start if it is crossed in backward direction)

3: forward-backward looping (change direction at both loop points if they are crossed as described above)

*ilps* (optional, defaults to 0) - loop start in sample frames (fractional values are allowed), or loop end if *ilps* is greater than *ilpe*. Ignored unless *ilpmod* is set to 1, 2, or 3. If the loop points are equal, the whole sample is looped.

*ilpe* (optional, defaults to 0) - loop end in sample frames (fractional values are allowed), or loop start if *ilps* is greater than *ilpe*. Ignored unless *ilpmod* is set to 1, 2, or 3. If the loop points are equal, the whole sample is looped.

## Credits

Written by Istvan Varga.

2006

New in Csound 5.03

# peakk

peakk — Deprecated.

## Description

Deprecated as of version 3.63. Use the *peak* opcode instead.

# pop

pop — Pops values from the global stack. Deprecated.

## Description

Pops values from the global stack.

## Syntax

```
xval1, [xval2, ... , xval31] pop  
ival1, [ival2, ... , ival31] pop
```

## Initialization

*ival1 ... ival31* - values to be popped from the stack.

## Performance

*xval1 ... xval31* - values to be popped from the stack.

The given values are popped from the stack. The global stack works in LIFO order: after multiple *push* calls, *pop* should be used in reverse order.

Each *push* or *pop* operation can work on a "bundle" of multiple variables. When using *pop*, the number, type, and order of items must match those used by the corresponding *push*. That is, after a 'push Sfoo, ibar', you must call something like 'Sbar, ifoo pop', and not e.g. two separate 'pop' statements.

*push* and *pop* opcodes can take variables of any type (i-, k-, a- and strings). Use of any combination of i, k, a, and S types is allowed. Variables of type 'a' and 'k' are passed at performance time only, while 'i' and 'S' are passed at init time only.

push/pop for a, k, i, and S types copy data by value. By contrast, *push\_f* only pushes a "reference" to the f-signal, and then the corresponding *pop\_f* will copy directly from the original variable to its output signal. For this reason, changing the source f-signal of *push\_f* before *pop\_f* is called is not recommended, and if the instrument instance owning the variable that was passed by *push\_f* is deactivated before *pop\_f* is called, undefined behavior may occur.

Any stack errors (trying to push when there is no more space, or pop from an empty stack, inconsistent number or type of arguments, etc.) are fatal and terminate performance.

## See also

## Examples

Here is an example of the pop opcode. It uses the file *pop.csd* [examples/pop.csd].

### Example 1230. Example of the pop opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o pop.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
Odbfs = 1
nchnls = 2

stack 100000

instr 1

a1 oscils 0.7, 220, 0
k1 line 0, p3, 1
    push "blah", 123.45, a1, k1
    push rnd(k1)

k_rnd pop
S01, i01, a01, k01 pop
    printf_i "S01 = '%s', i01 = %g\n", 1, S01, i01
ktrig metro 5.0
    printf "k01 = %.3f, k_rnd = %.3f\n", ktrig, k01, k_rnd
    outs a01, a01

endin
</CsInstruments>
<CsScore>

i 1 0 5
e
</CsScore>
</CsoundSynthesizer>
```

*stack, push, pop\_f and push\_f.*

Using this opcode is somewhat hackish, as you can read here: <http://csound.1045644.n5.nabble.com/passing-a-string-to-a-UDO-td1099284.html>

## Credits

By: Istvan Varga.

2006

Deprecated as of version 6.04.

# pop\_f

pop\_f — Pops an f-sig frame from the global stack. Deprecated.

## Description

Pops an f-sig frame from the global stack.

## Syntax

```
f sig pop_f
```

## Performance

*f sig* - f-signal to be popped from the stack.

The values are popped the stack. The global stack must be initialized before used, and its size must be set. The global stack works in LIFO order: after multiple *push\_f* calls, *pop\_f* should be used in reverse order.

push/pop for a, k, i, and S types copy data by value. By contrast, *push\_f* only pushes a "reference" to the f-signal, and then the corresponding *pop\_f* will copy directly from the original variable to its output signal. For this reason, changing the source f-signal of *push\_f* before *pop\_f* is called is not recommended, and if the instrument instance owning the variable that was passed by *push\_f* is deactivated before *pop\_f* is called, undefined behavior may occur.

*push\_f* and *pop\_f* can only take a single argument, and the data is passed both at init and performance time.

Any stack errors (trying to push when there is no more space, or pop from an empty stack, inconsistent number or type of arguments, etc.) are fatal and terminate performance.

## See also

*stack*, *push*, *pop* and *push\_f*.

## Credits

By: Istvan Varga.

2006

Deprecated as of version 6.04.

# push

`push` — Pushes a value into the global stack. Deprecated.

## Description

Pushes a value into the global stack.

## Syntax

```
push xval1, [xval2, ... , xval31]
```

```
push ival1, [ival2, ... , ival31]
```

## Initialization

*ival1 ... ival31* - values to be pushed into the stack.

## Performance

*xval1 ... xval31* - values to be pushed into the stack.

The given values are pushed into the global stack as a bundle. The global stack works in LIFO order: after multiple *push* calls, *pop* should be used in reverse order.

Each *push* or *pop* operation can work on a "bundle" of multiple variables. When using *pop*, the number, type, and order of items must match those used by the corresponding *push*. That is, after a 'push Sfoo, ibar', you must call something like 'Sbar, ifoo pop', and not e.g. two separate 'pop' statements.

*push* and *pop* opcodes can take variables of any type (i-, k-, a- and strings). Use of any combination of i, k, a, and S types is allowed. Variables of type 'a' and 'k' are passed at performance time only, while 'i' and 'S' are passed at init time only.

*push/pop* for a, k, i, and S types copy data by value. By contrast, *push\_f* only pushes a "reference" to the f-signal, and then the corresponding *pop\_f* will copy directly from the original variable to its output signal. For this reason, changing the source f-signal of *push\_f* before *pop\_f* is called is not recommended, and if the instrument instance owning the variable that was passed by *push\_f* is deactivated before *pop\_f* is called, undefined behavior may occur.

Any stack errors (trying to push when there is no more space, or pop from an empty stack, inconsistent number or type of arguments, etc.) are fatal and terminate performance.

## Examples

Here is an example of the *push* opcode. It uses the file *push.csd* [examples/push.csd].

### Example 1231. Example of the push opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o push.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 2

stack 100000

instr 1

a1 oscils 0.7, 220, 0
k1 line 0, p3, 1
    push "blah", 123.45, a1, k1
    push rnd(k1)

k_rnd pop
S01, i01, a01, k01 pop
    printf_i "S01 = '%s', i01 = %g\n", 1, S01, i01
ktrig metro 5.0
    printf "k01 = %.3f, k_rnd = %.3f\n", ktrig, k01, k_rnd
    outs a01, a01

endin
</CsInstruments>
<CsScore>

i 1 0 5
e
</CsScore>
</CsoundSynthesizer>
```

## See also

*stack*, *pop*, *pop\_f* and *push\_f*.

Using this opcode is somewhat hackish, as you can read here: <http://csound.1045644.n5.nabble.com/passing-a-string-to-a-UDO-td1099284.html>

## Credits

By: Istvan Varga.

2006

Deprecated as of version 6.04.

# push\_f

`push_f` — Pushes an f-sig frame into the global stack. Deprecated.

## Description

Pushes an f-sig frame into the global stack.

## Syntax

```
push_f fsig
```

## Performance

*fsig* - f-signal to be pushed into the stack.

The values are pushed into the global stack. The global stack works in LIFO order: after multiple *push\_f* calls, *pop\_f* should be used in reverse order.

push/pop for a, k, i, and S types copy data by value. By contrast, *push\_f* only pushes a "reference" to the f-signal, and then the corresponding *pop\_f* will copy directly from the original variable to its output signal. For this reason, changing the source f-signal of *push\_f* before *pop\_f* is called is not recommended, and if the instrument instance owning the variable that was passed by *push\_f* is deactivated before *pop\_f* is called, undefined behavior may occur.

*pop\_f* and *push\_f* can only take a single argument, and the data is passed both at init and performance time.

Any stack errors (trying to push when there is no more space, or pop from an empty stack, inconsistent number or type of arguments, etc.) are fatal and terminate performance.

## See also

*stack*, *push*, *pop* and *pop\_f*.

## Credits

By: Istvan Varga.

2006

Deprecated as of version 6.04.

# soundout

soundout — Deprecated. Writes audio output to a disk file.

## Description



### Note

The usage of *soundout* is discouraged. Please use *fout* instead.

Writes audio output to a disk file.

## Syntax

```
soundout  asig1, ifilcod [, iformat]
```

## Initialization

*ifilcod* -- integer or character-string denoting the destination soundfile name. An integer denotes the file soundin.filcod; a character-string (in double quotes, spaces permitted) gives the filename itself, optionally a full pathname. If not a full path, the named file is sought first in the current directory, then in that given by the environment variable *SSDIR* (if defined) then by *SFDIR*. See also *GEN01*.

*iformat* (optional, default=0) -- specifies the audio data file format:

- 1 = 8-bit signed char (high-order 8 bits of a 16-bit integer)
- 2 = 8-bit A-law bytes
- 3 = 8-bit U-law bytes
- 4 = 16-bit short integers
- 5 = 32-bit long integers
- 6 = 32-bit floats

If *iformat* = 0 it is taken from the soundfile header, and if no header from the Csound *-o* command-line flag. The default value is 0.

## Performance

*soundout* writes audio output to a disk file.



### Note

Use of *fout* is recommended instead of *soundout*

## See Also

*fout, out, outh, outo, outq, outq1, outq2, outq3, outq4, outs, outs1, outs2 soundouts*

## Credits

Author: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

# soundouts

soundouts — Deprecated. Writes audio output to a disk file.

## Description



### Note

The usage of *soundouts* is discouraged. Please use *fout* instead.

Writes audio output to a disk file.

## Syntax

```
soundouts asigl, asigr, ifilcod [, iformat]
```

## Initialization

*ifilcod* -- integer or character-string denoting the destination soundfile name. An integer denotes the file soundout.ifilcod; a character-string (in double quotes, spaces permitted) gives the filename itself, optionally a full pathname. If not a full path, the named file is written relative to the directory given by the SFDIR environment variable if defined, or the current directory. See also *GEN01*.

*iformat* (optional, default=0) -- specifies the audio data file format:

- 1 = 8-bit signed char (high-order 8 bits of a 16-bit integer)
- 4 = 16-bit short integers
- 5 = 32-bit long integers
- 6 = 32-bit floats

If *iformat* = 0 it is taken from the Csound *-o* command-line flag. The default value is 0.

## Performance

*soundouts* writes stereo audio output to a disk file in raw (headerless) format without 0dBFS scaling. The expected range of the audio signals depends on the selected sample format.



### Note

Use of *fout* is recommended instead of *soundouts*

## See Also

*out, outh, outo, outq, outq1, outq2, outq3, outq4, outs, outs1, outs2 soundout*

## Credits

Author: Istvan Varga

# stack

`stack` — Initializes the stack. Deprecated.

## Description

Initializes and sets the size of the global stack.

## Syntax

```
stack iStackSize
```

## Initialization

*iStackSize* - size of the stack in bytes.

## Performance

Csound implements a single global stack. Initializing the stack with the *stack* opcode is not required - it is optional, and if not done, the first use of *push* or *push\_f* will automatically create a stack of 32768 bytes. Otherwise, *stack* is normally called from the orchestra header, and takes a stack size parameter in bytes (there is an upper limit of about 16 MB). Once set, the stack size is fixed and cannot be changed during performance.

The global stack works in LIFO order: after multiple *push* calls, *pop* should be used in reverse order.

Each *push* or *pop* operation can work on a "bundle" of multiple variables. When using *pop*, the number, type, and order of items must match those used by the corresponding *push*. That is, after a 'push Sfoo, ibar', you must call something like 'Sbar, ifoo pop', and not e.g. two separate 'pop' statements.

*push* and *pop* opcodes can take variables of any type (i-, k-, a- and strings). Variables of type 'a' and 'k' are passed at performance time only, while 'i' and 'S' are passed at init time only.

push/pop for a, k, i, and S types copy data by value. By contrast, *push\_f* only pushes a "reference" to the f-signal, and then the corresponding *pop\_f* will copy directly from the original variable to its output signal. For this reason, changing the source f-signal of *push\_f* before *pop\_f* is called is not recommended, and if the instrument instance owning the variable that was passed by *push\_f* is deactivated before *pop\_f* is called, undefined behavior may occur.

Any stack errors (trying to push when there is no more space, or pop from an empty stack, inconsistent number or type of arguments, etc.) are fatal and terminate performance.

## Examples

Here is an example of the stack opcode. It uses the file *stack.csd* [examples/stack.csd].

### Example 1232. Example of the stack opcode.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```
<CsoundSynthesizer>
<CsOptions>
; Select audio/midi flags here according to platform
-odac   ;;realtime audio out
;-iadc   ;;uncomment -iadc if realtime audio input is needed too
; For Non-realtime ouput leave only the line below:
; -o stack.wav -W ;; for file output any platform
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
Odbfs = 1
nchnls = 2

stack 100000

instr 1

a1 oscils 0.7, 220, 0
k1 line 0, p3, 1
    push "blah", 123.45, a1, k1
    push rnd(k1)

k_rnd pop
S01, i01, a01, k01 pop
    printf_i "S01 = '%s', i01 = %g\n", 1, S01, i01
ktrig metro 5.0
    printf "k01 = %.3f, k_rnd = %.3f\n", ktrig, k01, k_rnd
    outs a01, a01

endin
</CsInstruments>
<CsScore>

i 1 0 5
e
</CsScore>
</CsoundSynthesizer>
```

## See also

*pop*, *push*, *pop\_f* and *push\_f*.

Using this opcode is somewhat hackish, as you can read here: <http://csound.1045644.n5.nabble.com/passing-a-string-to-a-UDO-td1099284.html>

## Credits

By: Istvan Varga.

2006

Deprecated as of version 6.04.

## Deprecated Gen Routines

The following GEN routine is deprecated. It is still distributed with Csound for backward compatibility. The new GEN routine which is to be used in place of the deprecated one is indicated within the following entry.

## GEN22

GEN22 — Deprecated.

### Description

Deprecated as of version 4.19. Use the *GEN18* routine instead.

---

# The Utility Programs

Dan Ellis, MIT Media Lab

The Csound Utilities are *soundfile preprocessing* programs that return information on a soundfile or create some analyzed version of it for use by certain Csound generators. Though different in goals, they share a common soundfile access mechanism and are describable as a set. The Soundfile Utility programs can be invoked in two equivalent forms:

```
csound [-U utilname] [flags] [filenames]
```

```
utilname [flags] [filenames]
```

In the first, the utility is invoked as part of the Csound executable, while in the second it is called as a standalone program. The second is smaller by about 200K, but the two forms are identical in function. The first is convenient in not requiring the maintenance and use of several independent programs - one program does all. When using this form, a *-U flag* detected in the command line will cause all subsequent flags and names to be interpreted as per the named utility; i.e. Csound generation will not occur, and the program will terminate at the end of utility processing.

## Directories.

Filenames are of two kinds, source soundfiles and resultant analysis files. Each has a hierarchical naming convention, influenced by the directory from which the Utility is invoked. Source soundfiles with a full pathname (begins with dot (.), slash (/), or for ThinkC includes a colon (:)), will be sought only in the directory named. Soundfiles without a path will be sought first in the current directory, then in the directory named by the *SSDIR* environment variable (if defined), then in the directory named by *SFDIR*. An unsuccessful search will return a "cannot open" error.

Resultant analysis files are written into the current directory, or to the named directory if a path is included. It is tidy to keep analysis files separate from sound files, usually in a separate directory known to the *SADIR* variable. Analysis is conveniently run from within the *SADIR* directory. When an analysis file is later invoked by a Csound generator it is sought first in the current directory, then in the directory defined by *SADIR*.

## Soundfile Formats.

Csound can read and write audio files in a variety of formats. Write formats are described by Csound command flags. On reading, the format is determined from the soundfile header, and the data automatically converted to floating-point during internal processing. When Csound is installed on a host with local soundfile conventions (SUN, NeXT, Macintosh) it may conditionally include local packaging code which creates soundfiles not portable to other hosts. However, Csound on any host can always generate and read AIFF files, which is thus a portable format. Sampled sound libraries are typically AIFF, and the variable *SSDIR* usually points to a directory of such sounds. If defined, the *SSDIR* directory is in the search path during soundfile access. Note that some AIFF sampled sounds have an audio looping feature for sustained performance; the analysis programs will traverse any loop segment once only.

For soundfiles without headers, an SR value may be supplied by the *-r flag* (or its default). If both the *SR header* and the command-line flag are present, the flag value will override the header.

When sound is accessed by the audio Analysis programs, only a single channel is read. For stereo or quad files, the default is channel one; alternate channels may be obtained on request.

## Analysis File Generation (ATSA, CVANAL, HETRO, LPANAL, PVANAL)

The following utilities exist for Soundfile analysis:

- *ATSA*: ATS analysis for use with the Csound *ATS Resynthesis* opcodes.
- *CVANAL*: Impulse Response Fourier Analysis for *convolve* operator.
- *HETRO*: Heterodyne analysis for the Csound *adsyn* generator.
- *LPANAL*: Linear predictive coding analysis for the Csound *Linear Predictive Coding (LPC) Resynthesis* opcodes.
- *PVANAL*: Phase vocoder analysis for the Csound *pvoc* generator.

## atsa

atsa — Performs ATS analysis on a soundfile.

### Description

ATS analysis for use with the Csound *ATS Resynthesis* opcodes.

### Syntax

```
csound -U atsa [flags] infilename outfilename
```

### Initialization

The following flags can be set for atsa (The default values are stated in parenthesis):

- b start (0.000000 seconds)
- e duration (0.000000 seconds or end)
- l lowest frequency (20.000000 Hertz)
- H highest frequency (20000.000000 Hertz)
- d frequency deviation (0.100000 of partial freq.)
- c window cycles (4 cycles)
- w window type (type: 1) (Options: 0=BLACKMAN, 1=BLACKMAN\_H, 2=HAMMING, 3=VON-HANN)
- h hop size (0.250000 of window size)
- m lowest magnitude (-60.000000)
- t track length (3 frames)
- s min. segment length (3 frames)
- g min. gap length (3 frames)
- T SMR threshold (30.000000 dB SPL)
- S min. segment SMR (60.000000 dB SPL)
- P last peak contribution (0.000000 of last peak's parameters)
- M SMR contribution (0.500000)
- F File Type (type: 4) (Options: 1=amp.and freq. only, 2=amp.,freq. and phase, 3=amp.,freq. and residual, 4=amp.,freq.,phase, and residual)

### Parameters

ATS analysis was devised by Juan Pampin. For complete information on ATS visit: <http://www-ccrma.s-tanford.edu/~juan/ATS.html>.

Analysis parameters must be carefully tuned for the Analysis Algorithm (ATSA) to properly capture the nature of the signal to be analyzed. As there are a significant number of them, ATSH offers the possibility of Saving/Loading them in a Binary File carrying the extension "\*.apf". The extension is not mandatory, but recommended. A brief explanation of each Analysis Parameters follows:

1. Start (secs.): the starting time of the analysis in seconds.
2. Duration (secs.): the duration time of the analysis in seconds. A zero means the whole duration of the input sound file.
3. Lowest Frequency (Hz.): this parameter will partially determine the size of the Analysis Window to be used. To compute the size of the Analysis Window, the period of the Lowest Frequency in samples (SR / LF) is multiplied by the number of cycles of it the user wants to fit in the Analysis Window (see parameter 6). This value is rounded to the next power of two to determine the size of the FFT for the analysis. The remaining samples are zero-padded. If the signal is a single, harmonic sound, then the

value of the Lowest Frequency should be its fundamental frequency or a sub-harmonic of it. If it is not harmonic, then its lowest significant frequency component may be a good starting value.

4. Highest Frequency (Hz.): highest frequency to be taken into account for Peak Detection. Once it is determined that no relevant information is found beyond a certain frequency, the analysis may be faster and more accurate setting the Highest Frequency parameter to that value.
5. Frequency Deviation (Ratio): frequency deviation allowed for each peak in the Peak Continuation Algorithm, as a ratio of the frequency involved. For instance, considering a peak at 440 Hz and a Deviation of .1 will produce that the Peak Continuation Algorithm will only try to find candidates for its continuation between 396 and 484 Hz (10% above and below the frequency of the peak). A small value is likely to produce more trajectories whilst a large value will reduce them, but at the cost of rendering information difficult to be further processed.
6. Number of Cycles of Lowest Frequency to fit in Analysis Window: this will also partially determine the size of the Fourier Analysis Window to be used. See Parameter 3. For single harmonic signals, it is supposed to be more than one (typically 4).
7. Hop Size (Ratio): size of the gap between one Analysis Window and the next expressed as a ratio of the Window Size. For instance, a Hop Size value of .25 will "jump" by 512 samples (Windows will overlap for a 75% of their size). This parameter will also determine the size of the analysis frames obtained. Signals that change their spectra very fast (such as Speech sounds) may need a high frame rate in order to properly track their changes.
8. Amplitude Threshold (dB): the highest amplitude value to be taken into account for Peak Detection.
9. Window Type: the shape of the smoothing function to be used for the Fourier Analysis. There are four choices available at present: Blackman, Blackman-Harris, Von Hann, and Hanning. Precise specifications about them are easily found on D.S.P. bibliography.
- 10.Track Length (Frames): The Peak Continuation Algorithm will "look-back" by Length frames in order to do its job better, preventing frequency trajectories from curving too much and loosing stability. However, a large value for this parameter will slow down the analysis significantly.
- 11.Minimal Segment Length (Frames): once the analysis is done, the spectral data can be further "cleaned" up during post-processing. Trajectories shorter than this value are suppressed if their average SMR is below Minimal Segment SMR (see parameters 16 and 14). This might help to avoid non-relevant sudden changes while keeping a high frame rate, reducing also the number of intermittent sinusoids during synthesis.
- 12.Minimal Gap Length (Frames): as parameter 11, this one is also used to clean up the data during post-processing. In this case, gaps (zero amplitude values, i.e. theoretical "silence") longer than Length frames are filled up with amplitude/frequency values obtained by linear interpolation of the adjacent active frames. This parameter prevents sudden interruptions of stable trajectories while keeping a high frame rate.
- 13.SMR Threshold (dB SPL): also a post-processing parameter, the SMR Threshold is used to eliminate partials with low averages.
- 14.Minimal Segment SMR (dB SPL): this parameter is used in combination with parameter 11. Short segments with SMR average below this value will be removed during post-processing.
- 15.Last Peak Contribution (0 to 1): as explained in Parameter 10, the Peak Continuation Algorithm "looks-back" several number of frames to do its job better. This parameter will help to weight the contribution of the first precedent peak over the others. A zero value means that all precedent peaks (to the size of Parameter 10) are equally taken in account.

16.SMR Contribution (0 to 1): In addition to the proximity in frequency of the peaks, the ATS Peak Continuation Algorithm may use psycho-acoustic information (the Signal-to-Mask-Ratio, or SMR) to improve the perceptual results. This parameter indicates how much the SMR information is used during tracking. For instance, a value of .5 makes the Peak Continuation Algorithm to use a 50% of SMR information and a 50% of Frequency Proximity information to decide which is the best candidate to continue a sinusoidal track.

## Examples

The following command:

```
atsa -b0.1 -e1 -l100 -H10000 -w2 audiofile.wav audiofile.ats
```

Generates the ATS analysis file 'audiofile.ats' from the original 'audiofile.wav' file. It begins analysis from second 0.1 of the file and the analysis is performed for 1 second thereafter. The lowest frequency stored is 100 Hz and the highest is 10kHz. A Hamming window is used for each analysis frame.

## cvanal

`cvanal` — Converts a soundfile into a single Fourier transform frame.

### Description

Impulse Response Fourier Analysis for *convolve* operator

### Syntax

```
csound -U cvanal [flags] infilename outfilename
```

```
cvanal [flags] infilename outfilename
```

### Initialization

`cvanal --` converts a soundfile into a single Fourier transform frame. The output file can be used by the *convolve* operator to perform Fast Convolution between an input signal and the original impulse response. Analysis is conditioned by the flags below. A space is optional between the flag and its argument.

`-s rate` -- sampling rate of the audio input file. This will over-ride the srate of the soundfile header, which otherwise applies. If neither is present, the default is 10000.

`-c channel` -- channel number sought. If omitted, the default is to process all channels. If a value is given, only the selected channel will be processed.

`-b begin` -- beginning time (in seconds) of the audio segment to be analyzed. The default is 0.0

`-d duration` -- duration (in seconds) of the audio segment to be analyzed. The default of 0.0 means to the end of the file.

`-X` -- write machine independent format analysis file.

### Examples

```
cvanal asound cvfile
```

will analyze the soundfile "asound" to produce the file "cvfile" for the use with *convolve*.

To use data that is not already contained in a soundfile, a soundfile converter that accepts text files may be used to create a standard audio file, e.g., the .DAT format for SOX. This is useful for implementing FIR filters.

### Files

The output file has a special *convolve* header, containing details of the source audio file. The analysis data is stored as "float", in rectangular (real/imaginary) form.



### Note

The analysis file is *not* system independent! Ensure that the original impulse recording/data is retained. If/when required, the analysis file can be recreated.

### Credits

Author: Greg Sullivan

Based on algorithm given in *Elements Of Computer Music*, by F. Richard Moore.

## hetro

hetro — Decomposes an input soundfile into component sinusoids.

### Description

Hetrodyne filter analysis for the Csound *adsyn* generator.

### Syntax

```
csound -U hetro [flags] infilename outfilename
```

```
hetro [flags] infilename outfilename
```

### Initialization

*hetro* takes an input soundfile, decomposes it into component sinusoids, and outputs a description of the components in the form of breakpoint amplitude and frequency tracks. Analysis is conditioned by the control flags below. A space is optional between flag and value.

*-s srate* -- sampling rate of the audio input file. This will over-ride the srate of the soundfile header, which otherwise applies. If neither is present, the default is 10000. Note that for *adsyn* synthesis the srate of the source file and the generating orchestra need not be the same.

*-c channel* -- channel number sought. The default is 1.

*-b begin* -- beginning time (in seconds) of the audio segment to be analyzed. The default is 0.0

*-d duration* -- duration (in seconds) of the audio segment to be analyzed. The default of 0.0 means to the end of the file. Maximum length is 32.766 seconds.

*-f begfreq* -- estimated starting frequency of the fundamental, necessary to initialize the filter analysis. The default is 100 (cps).

*-h partials* -- number of harmonic partials sought in the audio file. Default is 10, maximum is a function of memory available.

*-M maxamp* -- maximum amplitude summed across all concurrent tracks. The default is 32767.

*-m minamp* -- amplitude threshold below which a single pair of amplitude/frequency tracks is considered dormant and will not contribute to output summation. Typical values: 128 (48 db down from full scale), 64 (54 db down), 32 (60 db down), 0 (no thresholding). The default threshold is 64 (54 db down).

*-n brkpts* -- initial number of analysis breakpoints in each amplitude and frequency track, prior to thresholding (*-m*) and linear breakpoint consolidation. The initial points are spread evenly over the duration. The default is 256.

*-l cutfreq* -- substitute a 3rd order Butterworth low-pass filter with cutoff frequency *cutfreq* (in Hz), in place of the default averaging comb filter. The default is 0 (don't use).

### Performance

As of Csound 4.08, *hetro* can write SDIF output files if the output file name ends with ".sdif" or ".SDIF". See the *sdif2ad* utility for more information about the Csound's SDIF support.

### Examples

```
hetro -s44100 -b.5 -d2.5 -h16 -M24000 audiofile.test adsynfile7
```

This will analyze 2.5 seconds of channel 1 of a file "audiofile.test", recorded at 44.1 kHz, beginning .5 seconds from the start, and place the result in a file "adsynfile7". We request just the first 16 harmonics of the sound, with 256 initial breakpoint values per amplitude or frequency track, and a peak summation amplitude of 24000. The fundamental is estimated to begin at 100 Hz. Amplitude thresholding is at 54 db down.

The Butterworth LPF is not enabled.

## File Format

The output file contains time-sequenced amplitude and frequency values for each partial of an additive complex audio source. The information is in the form of breakpoints (time, value, time, value, ....) using 16-bit integers in the range 0 - 32767. Time is given in milliseconds, and frequency in Hertz (cps). The breakpoint data is exclusively non-negative, and the values -1 and -2 uniquely signify the start of new amplitude and frequency tracks. A track is terminated by the value 32767. Before being written out, each track is data-reduced by amplitude thresholding and linear breakpoint consolidation.

A component partial is defined by two breakpoint sets: an amplitude set, and a frequency set. Within a composite file these sets may appear in any order (amplitude, frequency, amplitude ....; or amplitude, amplitude..., then frequency, frequency,...). During *adsyn* resynthesis the sets are automatically paired (amplitude, frequency) from the order in which they were found. There should be an equal number of each.

A legal *adsyn* control file could have following format:

```
-1 time1 value1 ... timeK valueK 32767 ; amplitude breakpoints for partial 1
-2 time1 value1 ... timeL valueL 32767 ; frequency breakpoints for partial 1
-1 time1 value1 ... timeM valueM 32767 ; amplitude breakpoints for partial 2
-2 time1 value1 ... timeN valueN 32767 ; frequency breakpoints for partial 2
-2 time1 value1 .....
-2 time1 value1 ..... ; pairable tracks for partials 3 and 4
-1 time1 value1 .....
-1 time2 value1 .....
```

## Credits

Author: Tom Sullivan

1992

Author: John ffitc

1994

Author: Richard Dobson

2000

October 2002. Thanks to Rasmus Ekman, added a note about the SDIF format.

## lpanal

`lpanal` — Performs both linear predictive and pitch-tracking analysis on a soundfile.

### Description

Linear predictive analysis for the Csound *Linear Predictive Coding (LPC) Resynthesis* opcodes.

### Syntax

```
csound -U lpanal [flags] infilename outfilename
```

```
lpanal [flags] infilename outfilename
```

### Initialization

`lpanal` performs both lpc and pitch-tracking analysis on a soundfile to produce a time-ordered sequence of *frames* of control information suitable for Csound resynthesis. Analysis is conditioned by the control flags below. A space is optional between the flag and its value.

`-a -- [alternate storage]` asks `lpanal` to write a file with filter poles values rather than the usual filter coefficient files. When `lpread / lpreson` are used with pole files, automatic stabilization is performed and the filter should not get wild. (This is the default in the Windows GUI) - Changed by Marc Resibois.

`-s srate --` sampling rate of the audio input file. This will over-ride the srate of the soundfile header, which otherwise applies. If neither is present, the default is 10000.

`-c channel --` channel number sought. The default is 1.

`-b begin --` beginning time (in seconds) of the audio segment to be analyzed. The default is 0.0

`-d duration --` duration (in seconds) of the audio segment to be analyzed. The default of 0.0 means to the end of the file.

`-p npoles --` number of poles for analysis. The default is 34, the maximum 50.

`-h hopsize --` hop size (in samples) between frames of analysis. This determines the number of frames per second (`srate / hopsize`) in the output control file. The analysis framesize is `hopsize * 2` samples. The default is 200, the maximum 500.

`-C string --` text for the comments field of the lpcfile header. The default is the null string.

`-P mincps --` lowest frequency (in Hz) of pitch tracking. `-P0` means no pitch tracking.

`-Q maxcps --` highest frequency (in Hz) of pitch tracking. The narrower the pitch range, the more accurate the pitch estimate. The defaults are `-P70, -Q200`.

`-v verbosity --` level of terminal information during analysis.

- 0 = none
- 1 = verbose
- 2 = debug

The default is 0.

`-X --` write machine independent format analysis file.

## Examples

```
lpanal -a -p26 -d2.5 -P100 -Q400 audiofile.test lpfil22
```

will analyze the first 2.5 seconds of file "audiofile.test", producing  $\text{srate}/200$  frames per second, each containing 26-pole filter coefficients and a pitch estimate between 100 and 400 Hertz. Stabilized (*-a*) output will be placed in "lpfil22" in the current directory.

## File Format

Output is a file comprised of an identifiable header plus a set of frames of floating point analysis data. Each frame contains four values of pitch and gain information, followed by *npoles* filter coefficients. The file is readable by Csound's *lpread*.

*lpanal* is an extensive modification of Paul Lanksy's lpc analysis programs.

## pvanal

pvanal — Converts a soundfile into a series of short-time Fourier transform frames.

### Description

Fourier analysis for the Csound *pvoc* generator

### Syntax

```
csound -U pvanal [flags] infilename outfilename
```

```
pvanal [flags] infilename outfilename
```

### Pvanal extension to create a PVOC-EX file.

The standard Csound utility program pvanal has been extended to enable a PVOC-EX format file to be created, using the existing interface. To create a PVOC-EX file, the file name must be given the required extension, “.pvx”, e.g “test.pvx”. The requirement for the FFT size to be a power of two is here relaxed, and any positive value is accepted; odd numbers are rounded up internally. However, power-of-two sizes are still to be preferred for all normal applications.

The channel select flags are ignored, and all source channels will be analysed and written to the output file, up to a compiler-set limit of eight channels. The analysis window size (*iwinsize*) is set internally to double the FFT size.

### Initialization

*pvanal* converts a soundfile into a series of short-time Fourier transform (STFT) frames at regular time-points (a frequency-domain representation). The output file can be used by *pvoc* to generate audio fragments based on the original sample, with timescales and pitches arbitrarily and dynamically modified. Analysis is conditioned by the flags below. A space is optional between the flag and its argument.

*-s srate* -- sampling rate of the audio input file. This will over-ride the *srate* of the soundfile header, which otherwise applies. If neither is present, the default is 10000.

*-c channel* -- channel number sought. The default is 1.

*-b begin* -- beginning time (in seconds) of the audio segment to be analyzed. The default is 0.0

*-d duration* -- duration (in seconds) of the audio segment to be analyzed. The default of 0.0 means to the end of the file.

*-n frmsiz* -- STFT frame size, the number of samples in each Fourier analysis frame. Must be a power of two, in the range 16 to 16384. For clean results, a frame must be larger than the longest pitch period of the sample. However, very long frames result in temporal "smearing" or reverberation. The bandwidth of each STFT bin is determined by sampling rate / frame size. The default framesize is the smallest power of two that corresponds to more than 20 milliseconds of the source (e.g. 256 points at 10 kHz sampling, giving a 25.6 ms frame).

*-w windfact* -- Window overlap factor. This controls the number of Fourier transform frames per second. Csound's *pvoc* will interpolate between frames, but too few frames will generate audible distortion; too many frames will result in a huge analysis file. A good compromise for *windfact* is 4, meaning that each input point occurs in 4 output windows, or conversely that the offset between successive STFT frames is *framesize*/4. The default value is 4. Do not use this flag with *-h*.

*-h hopsiz* -- STFT frame offset. Converse of above, specifying the increment in samples between successive frames of analysis (see also *lpanal*). Do not use with *-w*.

*-H* -- Use a Hamming window instead of the default von Hann window.

*-K* -- Use a Kaiser window instead of the default von Hann window. The Kaiser parameter default is 6.8, but can be set with the *-B* option.

*-B beta* -- Set the beta parameter for any Kaiser window used to the floating point value beta.

## Examples

```
pvanal asound pvfile
```

will analyze the soundfile "asound" using the default frmsiz and windfact to produce the file "pvfile" suitable for use with pvoc.

## Files

The output file has a special *pvoc* header containing details of the source audio file, the analysis frame rate and overlap. Frames of analysis data are stored as float, with the magnitude and "frequency" (in Hz) for the first  $N/2 + 1$  Fourier bins of each frame in turn. "Frequency" encodes the phase increment in such a way that for strong harmonics it gives a good indication of the true frequency. For low amplitude or rapidly moving harmonics it is less meaningful.

## Diagnostics

Prints total number of frames, and frames completed every 20th frame.

## Credits

Author: Dan Ellis

MIT Media Lab

Cambridge, Massachussetts

1990

## File Queries (SNDINFO)

The following utilities exist for Soundfile query:

- *SNDINFO*: Displays information about a soundfile.

## sndinfo

`sndinfo` — Displays information about a soundfile.

### Description

Get basic information about one or more soundfiles.

### Syntax

```
csound -U sndinfo [options] soundfilenames ...  
sndinfo [options] soundfilenames ...
```

### Initialization

`sndinfo` will attempt to find each named file, open it for reading, read in the soundfile header, then print a report on the basic information it finds. The order of search across soundfile directories is as described above. If the file is of type AIFF, some further details are listed first.

There are two option types:

1. `-i` or `-il` will print instrument information, which includes looping. The option continues until a `-i0` option.
2. The other option is `-b` which prints the broadcast information for WAV files. It can similarly be negated with `-b0`.

### Examples

```
csound -U sndinfo test Bosendorfer/"BOSEN mf A0 st" foo foo2
```

where the environment variables `SFDIR = /u/bv/sound`, and `SSDIR = /so/bv/Samples`, might produce the following:

```
util SNDINFO:
/u/bv/sound/test:
  srate 22050, monaural, 16 bit shorts, 1.10 seconds
  headersiz 1024, datasiz 48500 (24250 sample frames)

/so/bv/Samples/Bosendorfer/BOSEN mf A0 st: AIFF, 197586 stereo samples, base Frq 261.6 (MIDI 60),
AIFF soundfile, looping with modes 1, 0
  srate 44100, stereo, 16 bit shorts, 4.48 seconds

  headersiz 402, datasiz 790344 (197586 sample frames)

/u/bv/sound/foo:
  no recognizable soundfile header

/u/bv/sound/foo2:
  couldn't find
```

## File Conversion (HET\_IMPORT, HET\_EXPORT, PVLOOK, PV\_EXPORT, PV\_IMPORT, SDIF2AD, SRCONV)

The following utilities exist for file conversion:

- `HET_EXPORT`: Exports a .het (produced by `HETRO`) to a comma separated text file.

- *HET\_IMPORT*: Generates a .het (in the format produced by *HETRO*) from a comma separated text file for usage with the *adsyn* generator.
- *PVLOOK*: View formatted text output of STFT analysis files.
- *PV\_EXPORT*: Converts a file generated by *PVANAL* to a text file.
- *PV\_IMPORT*: Converts a text file (in the format generated by *PV\_EXPORT*) to a *PVANAL* format file to be used with the *pvoc* opcode.
- *SDIF2AD*: Converts SDIF files to files usable by *adsynt*.
- *SRCONV*: Converts the sample rate of an audio file.

## dnoise

dnoise — Reduces noise in a file.

### Description

This is a noise reduction scheme using frequency-domain noise-gating.

### Syntax

```
dnoise [flags] -i noise_ref_file -o output_soundfile input_soundfile
```

### Initialization

Dnoise specific flags:

- *(no flag)* input soundfile to be denoised
- *-i fname* input reference noise soundfile
- *-o fname* output soundfile
- *-N fnum* # of bandpass filters (default: 1024)
- *-w fovlp* filter overlap factor: {0,1,(2),3} DON'T USE *-w* AND *-M*
- *-M awlen* analysis window length (default: N-1 unless *-w* is specified)
- *-L swlen* synthesis window length (default: M)
- *-D dfac* decimation factor (default: M/8)
- *-b btim* begin time in noise reference soundfile (default: 0)
- *-B smpst* starting sample in noise reference soundfile (default: 0)
- *-e etim* end time in noise reference soundfile (default: end of file)
- *-E smpend* final sample in noise reference soundfile (default: end of file)
- *-t thr* threshold above noise reference in dB (default: 30)
- *-S gfact* sharpness of noise-gate turnoff, range: 1 to 5 (default: 1)
- *-n numfrm* number of FFT frames to average over (default: 5)
- *-m mingain* minimum gain of noise-gate when off in dB (default: -40)

Soundfile format options:

- *-A* AIFF format output
- *-W* WAV format output
- *-J* IRCAM format output
- *-h* skip soundfile header (not valid for AIFF/WAV output)
- *-8* 8-bit unsigned char sound samples

- *-c* 8-bit signed\_char sound samples
- *-a* alaw sound samples
- *-u* ulaw sound samples
- *-s* short\_int sound samples
- *-l* long\_int sound samples
- *-f* float sound samples. Floats also supported for WAV files. (New in Csound 3.47.)

Additional options:

- *-R* verbose - print status info
- *-H [N]* print a heartbeat character at each soundfile write.
- *-- fname* output to log file fname
- *-V* verbose - print status info



### Note

DNOISE also looks at the environment variable SFOUTYP to determine soundfile output format.

The *-i* flag is used for a reference noise file (normally created from a short section of the denoised file, where only noise is audible). The input soundfile to be denoised can be given anywhere on the command line, without a flag.

## Performance

This is a noise reduction scheme using frequency-domain noise-gating. This should work best in the case of high signal-to-noise with hiss-type noise.

The algorithm is that suggested by Moorer & Berger in “Linear-Phase Bandsplitting: Theory and Applications” presented at the 76th Convention 1984 October 8-11 New York of the Audio Engineering Society (preprint #2132) except that it uses the Weighted Overlap-Add formulation for short-time Fourier analysis-synthesis in place of the recursive formulation suggested by Moorer & Berger. The gain in each frequency bin is computed independently according to

$$\text{gain} = g0 + (1-g0) * [\text{avg} / (\text{avg} + \text{th}*\text{th}*nref)] ^ \text{sh}$$

where *avg* and *nref* are the mean squared signal and noise respectively for the bin in question. (This is slightly different than in Moorer & Berger.)

The critical parameters *th* and *g0* are specified in dB and internally converted to decimal values. The *nref* values are computed at the start of the program on the basis of a noise\_soundfile (specified in the command line) which contains noise without signal.

The *avg* values are computed over a rectangular window of *m* FFT frames looking both ahead and behind the current time. This corresponds to a temporal extent of *m\*D/R* (which is typically *(m\*N/8)/R*). The default settings of *N*, *M*, and *D* should be appropriate for most uses. A higher sample rate than 16 KHz might indicate a higher *N*.

## Credits

Author: Mark Dolson

August 26, 1989

Author: John ffitch

December 30, 2000

Updated by Rasmus Ekman on March 11, 2002.

## het\_export

`het_export` — Converts a .het file to a comma separated text file.

### Syntax

```
het_export het_file ctext_file  
csound -U het_export het_file ctext_file
```

### Initialization

*het\_file* - Name of the input .het file.

*ctext\_file* - Name of the output comma-separated text file.

The *het\_export* utility generates a comma-separated text file for manual editing of a .het file produced by the *HETRO* utility. It can be used in combination with *het\_import* to produce data for the *adsyn* generator.

### Credits

Author: John ffitth

1995

## het\_import

`het_import` — Converts a comma-separated text file to a .het file.

### Syntax

```
het_import ctext_file het_file  
csound -U het_import ctext_file het_file
```

### Initialization

*ctext\_file* - Name of the input comma-separated text file.

*het\_file* - Name of the output .het file.

The *het\_import* utility generates a *.het* file usable with the *adsyn* generator. It can be used in combination with *het\_export* to modify sound analysis made by the *HETRO* utility.

### Credits

Author: John ffitch

1995







The Utility Programs

0.245 0.245 0.246 0.249 0.251 0.250 0.249 0.247 0.244 0.246  
0.248 0.250 0.252 0.250 0.247 0.246 0.245 0.247 0.249 0.250  
0.251 0.249 0.246 0.246 0.245 0.247 0.250 0.250 0.250 0.249  
0.245 0.245 0.246 0.248 0.251 0.251 0.249 0.248 0.245 0.245  
0.247 0.249 0.251 0.251 0.248 0.246 0.245 0.245 0.248 0.250  
0.251 0.250 0.247 0.245 0.245 0.246 0.249 0.251 0.250 0.249  
0.246 0.244 0.246 0.247 0.250 0.251 0.250 0.248 0.246 0.245  
0.247 0.249 0.250 0.251 0.249 0.247 0.246 0.245 0.247 0.250  
0.250 0.251 0.248 0.245 0.245 0.246 0.248 0.251 0.251 0.249  
0.248 0.245 0.245 0.247 0.249 0.251 0.251 0.248 0.247 0.245  
0.245 0.248 0.249 0.250 0.250 0.247 0.246 0.246 0.246 0.249  
0.251 0.250 0.250 0.246 0.245 0.246 0.247 0.250 0.251 0.249  
0.248 0.246 0.244 0.246 0.248 0.250 0.251 0.249 0.247 0.246  
0.245 0.247 0.250 0.250 0.251 0.249 0.245 0.245 0.246 0.248  
0.251 0.250 0.250 0.248 0.245 0.245 0.247 0.248 0.251 0.250  
0.248 0.247 0.245 0.246 0.248 0.250 0.251 0.250 0.247 0.246  
0.245 0.246 0.249 0.251 0.250 0.249 0.246 0.245 0.246 0.247  
0.250 0.251 0.250 0.249 0.246 0.244 0.246 0.248 0.250 0.251  
0.249 0.247 0.246 0.245 0.247 0.249 0.250 0.251 0.287 0.331  
0.178 0.008 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000  
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000  
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000  
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000  
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000  
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000  
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000  
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000  
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000  
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000  
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000  
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000  
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000  
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000  
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000  
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000  
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000  
0.140 1.265 2.766 3.289 3.296 3.293 3.296 3.296 3.296 3.290 3.293  
3.292 3.291 3.297 3.295 3.294 3.296 3.291 3.292 3.294 3.291  
3.296 3.297 3.292 3.295 3.292 3.290 3.295 3.293 3.294 3.297  
3.292 3.293 3.294 3.290 3.295 3.295 3.292 3.296 3.293 3.291  
3.294 3.291 3.293 3.297 3.292 3.295 3.294 3.288 3.293 3.293  
3.292 3.297 3.294 3.292 3.295 3.290 3.292 3.295 3.292 3.295  
3.295 3.290 3.294 3.292 3.292 3.292 3.297 3.293 3.293 3.295 3.290  
3.292 3.293 3.290 3.296 3.296 3.292 3.295 3.291 3.290 3.294  
3.291 3.294 3.296 3.291 3.293 3.293 3.290 3.295 3.294 3.293  
3.296 3.291 3.291 3.293 3.290 3.294 3.296 3.292 3.295 3.293  
3.288 3.293 3.292 3.292 3.297 3.292 3.293 3.294 3.289 3.292  
3.294 3.291 3.296 3.293 3.291 3.294 3.291 3.292 3.296 3.292  
3.294 3.295 3.289 3.292 3.292 3.291 3.296 3.294 3.292 3.295  
3.290 3.290 3.293 3.291 3.295 3.296 3.291 3.294 3.291 3.289  
3.294 3.292 3.293 3.295 3.291 3.292 3.293 3.290 3.294 3.295  
3.292 3.294 3.291 3.289 3.293 3.291 3.293 3.296 3.292 3.293  
3.293 3.288 3.292 3.293 3.292 3.296 3.293 3.291 3.294 3.289  
3.292 3.295 3.291 3.294 3.293 3.289 3.292 3.291 3.290 3.295  
3.293 3.292 3.294 3.289 3.291 3.293 3.290 3.295 3.294 3.290  
3.293 3.290 3.289 3.294 3.291 3.293 3.295 3.290 3.292 3.292  
3.289 3.293 3.293 3.292 3.295 3.291 3.289 3.292 3.290 3.292  
3.295 3.291 3.293 3.292 3.288 3.292 3.291 3.291 3.295 3.291  
3.291 3.292 3.289 3.291 3.294 3.291 3.294 3.292 3.289 3.292  
3.290 3.290 3.295 3.292 3.293 3.294 3.289 3.291 3.292 3.290  
3.294 3.293 3.291 3.293 3.289 3.290 3.293 3.291 3.294 3.295  
3.290 3.292 3.291 3.289 3.294 3.293 3.292 3.294 3.290 3.290  
3.292 3.289 3.293 3.294 3.291 3.293 3.291 3.289 3.292 3.291  
3.291 3.295 3.291 3.291 3.292 3.288 3.292 3.293 3.291 3.295  
3.292 3.290 3.292 3.289 3.291 3.294 3.291 3.293 3.292 3.288  
3.291 3.291 3.290 3.295 3.292 3.291 3.293 3.289 3.290 3.292  
3.290 3.294 3.293 3.290 3.292 3.290 3.289 3.293 3.291 3.292  
3.294 3.290 3.290 3.291 3.289 3.293 3.293 3.291 3.293 3.290  
3.288 3.291 3.290 3.292 3.294 3.290 3.292 3.291 3.288 3.291  
3.291 3.291 3.294 3.291 3.290 3.291 3.288 3.291 3.293 3.291  
3.293 3.292 3.288 3.291 3.290 3.290 3.294 3.291 3.291 3.292

3.288 3.290 3.291 3.290 3.294 3.293 3.290 3.292 3.289 3.289  
3.293 3.290 3.292 3.293 3.289 3.291 3.290 3.289 3.293 3.292  
3.291 3.293 3.289 3.289 3.291 3.289 3.292 3.293 3.290 3.292  
3.290 3.288 3.292 3.291 3.291 3.294 3.290 3.290 3.291 3.288  
3.291 3.292 3.291 3.293 3.291 3.288 3.291 3.289 3.290 3.293  
3.290 3.292 3.292 3.288 3.291 3.291 3.290 3.293 3.291 3.290  
3.292 3.288 3.289 3.292 3.290 3.292 3.293 3.289 3.291 3.289  
3.288 3.293 3.291 3.292 3.292 3.288 3.289 3.290 3.288 3.292  
3.293 3.290 3.292 3.289 3.288 3.291 3.290 3.291 3.293 3.289  
3.290 3.290 3.287 3.291 3.291 3.290 3.293 3.290 3.288 3.290  
3.288 3.290 3.293 3.291 3.292 3.291 3.288 3.290 3.289 3.289  
3.293 3.290 3.290 3.291 3.287 3.289 3.291 3.289 3.292 3.291  
3.288 3.290 3.288 3.288 3.292 3.290 3.291 3.292 3.288 3.289  
3.290 3.288 3.292 3.292 3.290 3.292 3.289 3.288 3.291 3.289  
3.291 3.293 3.289 3.291 3.290 3.287 3.291 3.290 3.290 3.293  
3.289 3.289 3.290 3.287 3.290 3.292 3.290 3.292 3.290 3.287  
3.290 3.289 3.289 3.292 3.290 3.290 3.291 3.287 3.289 3.290  
3.289 3.292 3.291 3.289 3.291 3.288

*etc...*

## Credits

Author: Richard Karpen

Seattle, Wash

1993 (New in Csound version 3.57)

## **pv\_export**

`pv_export` — Converts a .pvx file to a comma separated text file.

### **Syntax**

```
pv_export pv_file ctext_file  
csound -U pv_export pv_file ctext_file
```

### **Initialization**

*pv\_file* - Name of the input .pvx file.

*ctext\_file* - Name of the output comma-separated text file.

The *pv\_export* utility generates a comma-separated text file for manual editing of a .pvx file produced by the *PVANAL* utility. It can be used in combination with *pv\_import* to produce data for the *pvoc* generator.

### **Credits**

Author: John ffitth

1995

## **pv\_import**

`pv_import` — Converts a comma-separated text file to a .pvx file.

### **Syntax**

```
pv_import ctext_file pv_file  
csound -U pv_import ctext_file pv_file
```

### **Initialization**

*ctest\_file* - Name of the input comma-separated text file.

*pv\_file* - Name of the output .pvx file.

The *pv\_import* utility generates a .pvx file usable with the *pvoc* generator. It can be used in combination with *pv\_export* to modify sound analysis made by the *PVANAL* utility.

### **Credits**

Author: John fitch

1995

## sdif2ad

sdif2ad — Converts SDIF files to files usable by adsyn.

### Description

Convert files Sound Description Interchange Format (SDIF) to the format usable by Csound's *adsyn* opcode. As of Csound version 4.10, *sdif2ad* was available only as a standalone program for Windows console and DOS.

### Syntax

```
sdif2ad [flags] infilename outfilename
```

### Initialization

Flags:

- *-sN* -- apply amplitude scale factor N
- *-pN* -- keep only the first N partials. Limited to 1024 partials. The source partial track indices are used directly to select internal storage. As these can be arbitrary values, the maximum of 1024 partials may not be realized in all cases.
- *-r* -- byte-reverse output file data. The byte-reverse option is there to facilitate transfer across platforms, as Csound's *adsyn* file format is not portable.

If the filename passed to *hetro* has the extension “.sdif”, data will be written in SDIF format as 1TRC frames of additive synthesis data. The utility program *sdif2ad* can be used to convert any SDIF file containing a stream of 1TRC data to the Csound *adsyn* format. *sdif2ad* allows the user to limit the number of partials retained, and to apply an amplitude scaling factor. This is often necessary, as the SDIF specification does not, as of the release of *sdif2ad*, require amplitudes to be within a particular range. *sdif2ad* reports information about the file to the console, including the frequency range.

The main advantages of SDIF over the *adsyn* format, for Csound users, is that SDIF files are fully portable across platforms (data is “big-endian”), and do not have the duration limit of 32.76 seconds imposed by the 16 bit *adsyn* format. This limit is necessarily imposed by *sdif2ad*. Eventually, SDIF reading will be incorporated directly into *adsyn*, thus enabling files of any length (subject to system memory limits) to be analysed and processed.

Users should remember that the SDIF formats are still under development. While the 1TRC format is now fairly well established, it can still change.

For detailed information on the Sound Description Interchange Format, refer to the CNMAT website: <http://cnmat.CNMAT.Berkeley.EDU/SDIF>

Some other SDIF resources (including a viewer) are available via the NC\_DREAM website: <http://www.bath.ac.uk/~masjpf/NCD/dreamhome.html>

### Credits

Author: Richard Dobson

Somerset, England

August, 2000

New in Csound version 4.08

## srconv

srconv — Converts the sample rate of an audio file.

### Description

Converts the sample rate of an audio file at sample rate  $R_{in}$  to a sample rate of  $R_{out}$ . Optionally the ratio ( $R_{in} / R_{out}$ ) may be linearly time-varying according to a set of (time, ratio) pairs in an auxiliary file.



### Note

This utility is suspect. Better is to use libamplerate. srconv will be removed or replaced soon.

### Syntax

```
srconv [flags] infile
```

### Initialization

Flags:

- *-P num* = pitch transposition ratio (srate / r) [don't specify both P and r]
- *-Q num* = quality factor (1, 2, 3, or 4: default = 2)
- *-i filnam* = auxiliary breakpoints file (no breakpoint by default. i.e. No ratio change)
- *-r num* = output sample rate (must be specified)
- *-o fnam* = sound output filename
- *-A* = create an AIFF format output soundfile
- *-J* = create an IRCAM format output soundfile
- *-W* = create a WAV format output soundfile
- *-h* = no header on output soundfile
- *-c* = 8-bit signed\_char sound samples
- *-a* = alaw sound samples
- *-8* = 8-bit unsigned\_char sound samples
- *-u* = ulaw sound samples
- *-s* = short\_int sound samples
- *-l* = long\_int sound samples
- *-f* = float sound samples
- *-r N* = orchestra srate override
- *-K* = Do not generate PEAK chunks
- *-R* = continually rewrite header while writing soundfile (WAV/AIFF)

- *-H#* = print a heartbeat style 1, 2 or 3 at each soundfile write
- *-N* = notify (ring the bell) when score or miditrack is done
- *--fnam* = log output to file

This program performs arbitrary sample-rate conversion with high fidelity. The method is to step through the input at the desired sampling increment, and to compute the output points as appropriately weighted averages of the surrounding input points. There are two cases to consider:

1. sample rates are in a small-integer ratio - weights are obtained from table.
2. sample rates are in a large-integer ratio - weights are linearly interpolated from table.

Calculate increment: if decimating, then window is impulse response of low-pass filter with cutoff frequency at half of output sample rate; if interpolating, then window is impulse response of lowpass filter with cutoff frequency at half of input sample rate.

## Credits

Author: Mark Dolson

August 26, 1989

Author: John ffitc

December 30, 2000

## Other Csound Utilities (CS, CSB64ENC, ENVEXT, EXTRACTOR, MAKECSD, MIXER, SCALE, MKDB)

The following miscellaneous utilities are available:

- *CS*: Starts Csound with a set of options that can be controlled by environment variables, and input and output files determined by the specified filename stem.
- *CSB64ENC*: Converts a binary file to a Base64 encoded text file.
- *ENVEXT*: Extract the envelope of a file to a text list.
- *EXTRACTOR*: Extract a section of audio from an audio file.
- *MAKECSD*: Creates a CSD file from the specified input files.
- *MIXER*: Mixes together a number of soundfiles.
- *SCALE*: Scale the amplitude of a sound file.
- *MKDB*: Creates a plugin opcode catalog.

**CS**

**cs** — Starts Csound with a set of options that can be controlled by environment variables, and input and output files determined by the specified filename stem.

**Description**

Starts Csound with a set of options that can be controlled by environment variables, and input and output files determined by the specified filename stem.

**Syntax**

```
cs [-OPTIONS] <name> [CSOUND OPTIONS ... ]
```

**Initialization**

Flags:

- *-OPTIONS* = OPTIONS is a sequence of alphabetic characters that can be used for selecting the Csound executable to be run, as well as the command line flags (see below). There is a default for the option 'r' (selects real-time output), but it can be overridden.
- *<name>* = this is the filename stem for selecting input files; it may contain a path. Files that have .csd, .orc, or .sco extension are searched, and either a CSD or an orc/sco pair that matches *<name>* the best are selected. MIDI files with a .mid extension are also searched, and if one that matches *<name>* at least as close as the CSD or orc/sco pair, it is used with the -F flag.

**NOTE**

The MIDI file is not used if any -M or -F flag is specified by the user - new in version 4.24.0) Unless there is any option (-n or -o) related to audio output, an output file name with the appropriate extension is automatically generated (based on the name of selected input files and format options). The output file is always written to the current directory.

**NOTE**

file name extensions are not case sensitive.

- [*CSOUND OPTIONS ...*] = any number of additional options for Csound that are simply copied to the final command line to be executed.

The command line that is executed is generated from four parts:

1. Csound executable (possibly with options). This is exactly one of the following (the last one has the highest precedence):
  - a built-in default
  - the value of the CSOUND environment variable
  - environment variables with a name in the format of CSOUND\_x where x is an uppercase letter selected by characters of the -OPTIONS string. Thus, if the -dcba option is used, and the environment variables CSOUND\_B and CSOUND\_C are defined, the value of CSOUND\_B will take effect.
2. Any number of option lists, added in the following order:
  - either some built-in defaults, or the value of the CSFLAGS environment variable if it is defined.

- environment variables with a name in the format of CSFLAGS\_x where x is an uppercase letter selected by characters of the -OPTIONS string. Thus, if the -dcba option is used, and the environment variables CSFLAGS\_A and CSFLAGS\_C are defined as '-M 1 -o dac' and '-m231 -H0', respectively, the string '-m231 -H0 -M 1 -o dac' will be added.

3. The explicit options of [CSOUND OPTIONS ... ].
4. Any options and file names generated from <name>.



## NOTE

Quoted options that contain spaces are allowed.

## Examples

Assuming the following environment variables:

```
CSOUND      = csoundfltk.exe -W
CSOUND_D    = csound64.exe -J
CSOUND_R    = csoundfltk.exe -h

CSFLAGS     = -d -m135 -H1 -s
CSFLAGS_D   = -f
CSFLAGS_R   = -m0 -H0 -o dac1 -M "MIDI Yoke NT: 1" -b 200 -B 6000
```

And a directory that contains:

```
foo.orc          piano.csd
foo.sco          piano.mid
im.csd           piano2.mid
ImproSculpt2_share.csd foobar.csd
```

The following commands will execute as shown:

```
cs foo          => csoundfltk.exe -W -d -m135 -H1 -s -o foo.wav \
foo.orc foo.sco

cs foob         => csoundfltk.exe -W -d -m135 -H1 -s          \
-o foobar.wav foobar.csd

cs -r imp -i adc => csoundfltk.exe -h -d -m135 -H1 -s -m0 -H0 \
-o dac1 -M "MIDI Yoke NT: 1" \
-b 200 -B 6000 -i adc \
ImproSculpt2_share.csd

cs -d im        => csound64.exe -J -d -m135 -H1 -s -f -o im.sf \
im.csd

cs piano        => csoundfltk.exe -W -d -m135 -H1 -s          \
-F piano.mid -o piano.wav \
piano.csd

cs piano2       => csoundfltk.exe -W -d -m135 -H1 -s          \
-F piano2.mid -o piano2.wav \
piano.csd
```

## Credits

Author: Istvan Varga

Jan 2003

## csb64enc

`csb64enc` — Converts a binary file to a Base64 encoded text file.

### Description

The *csb64enc* utility generates a Base64 encoded text file from a binary file, such as a standard MIDI file (.mid) or any type of audio file. It is useful to convert a file in the format accepted by the `<CsFileB>` section of a csd file, to include the file within it.

### Syntax

```
csb64enc [OPTIONS ... ] infile1 [ infile2 [ ... ]]
```

### Initialization

Flags:

- `-w n` = set line width of the output file to `n` (default: 72)
- `-o fname` = output file name (default: stdout)

### Examples

```
csb64enc -w 78 -o file.txt file.mid
```

This command produces a Base64 encoded text file from the standard MIDI file *file.mid*. This file can now be pasted within a csd file's `<CsFileB>` section.

### See also

*makecsd*

### Credits

Author: Istvan Varga

Jan 2003

## envext

envext — Extracts the envelope of a file to a text file.

### Syntax

```
envext [-flags] soundfile
csound -U envext [-flags] soundfile
```

### Initialization

*soundfile* - Name of the input soundfile.

The following flags are available for envext (The default values are stated in parenthesis):

*-o fnam* Name of output filename (newenv)  
*-w size* (in seconds) of analysis window (0.25)

The *envext* utility generates a text file containing time and amplitude pairs by finding the absolute peak within each window.

### Example

Using the command (while in the manual directory):

```
csound -U envext examples/mary.wav
```

will produce the a text file containing the following:

```
0.000 0.000
0.000 0.000
0.250 0.000
0.500 0.000
0.750 0.000
1.249 0.170
1.499 0.269
1.530 0.307
1.872 0.263
2.056 0.304
2.294 0.241
2.570 0.216
2.761 0.178
3.077 0.011
3.251 0.001
3.500 0.000
```

Which shows the time for the peak amplitude within each measured window.

### Credits

Author: John ffitc

1995

## extractor

extractor — Extract a section of audio from an audio file.

### Description

Extract a section of audio, by time or sample, from an existing sound file.

### Syntax

```
extractor [OPTIONS ... ] infile
```

### Initialization

Flags:

- *-S integer* = Start the extract at the given sample number.
- *-Z integer* = End the extract at the given sample number.
- *-Q integer* = Extract given number of samples.
- *-T fptime* = Start the extract at the given time in seconds.
- *-E fptime* = End the extract at the given time in seconds.
- *-D fptime* = Extract given time in seconds.
- *-v* = Verbose mode.
- *-R* = Continually rewrite the header while writing soundfile (WAV/AIFF).
- *-H integer* = Show a "heart-beat" to indicate progress, in style 1, 2 or 3.
- *-N* = Alert call (usually ringing the bell) when finished.
- *-v* = Verbose mode.
- *-o fname* = output file name (default: test.wav)

### Examples

The default values are

```
extractor -S 0 -Z end-of-file -otest
```

For example

```
extractor -S 10234 -D 2.13 in.aiff -o out.wav
```

This creates a new sound file taken from sample 10234 and lasting 2.13 seconds.

### Credits

Author: John ffitth

1994

## makecsd

makecsd — Creates a CSD file from the specified input files.

### Description

Creates a CSD file from the specified input files. The first input file that has a .orc extension (case is not significant) is put to the <CsInstruments> section, and the first input file that has a .sco extension becomes <CsScore>. Any remaining files are Base64 encoded and added as <CsFileB> tags. An empty <CsOptions> section is always added.

Some text filtering is performed on the orchestra and score file:

- newlines are converted to the native format of the system on which makecsd is being run.
- blank lines are removed from the beginning and end of files.
- any trailing whitespace is removed from the end of lines.
- optionally, tabs can be expanded to spaces with an user specified tabstop size.
- optionally, a MIDI file can be included.
- optionally, a Licence can be specified as either a file or a common one.
- if required a score processor can be specified for the <CsScore> section.

### Syntax

```
makecsd [OPTIONS ... ] infile1 [ infile2 [ ... ]]
```

### Initialization

Flags:

- - *t n* = expand tabs to spaces using tabstop size *n* (default: disabled). This applies only to the orchestra and score file.
- - *w n* = set Base64 line width to *n* (default: 72). Note: the orchestra and score are not wrapped.
- - *o fname* = output file name (default: stdout)
- - *m fname* = specify a MIDI file to include (default: none)
- - *b progame* = specify the program to process the score (default: none)
- - *L fname* = file name containing the licence text (default: none)
- - *l integer* = specify a standard licence (default: none). The ones available are:
  - 0: All rights reserved
  - 1: CC BY-NC-ND
  - 2: CC BY-NC-SA
  - 3: CC BY-NC

- 4: CC BY-ND
- 5: CC BY-SA
- 6: CC BY
- 7: Licenced under BSD

## Examples

```
makecsd -t 6 -w 78 -o file.csd file.mid file.orc file.sco sample.aif
```

This creates a CSD from file.orc and file.sco (tabs are expanded to spaces assuming a tabstop size of 6 characters), and file.mid and sample.aif are added as <CsFileB> tags containing Base64 encoded data with a line width of 78 characters. The output file is file.csd.

## Credits

Author: Istvan Varga

Jan 2003

Author: John ffitc

Feb 2011

Options for MIDI, score processing and licence new in version 5.14

## mixer

mixer — Mixes together a number of soundfiles.

### Description

Mixes together a number of soundfiles, starting at different times and with individual channel selection from the input files.

### Syntax

```
mixer [OPTIONS ... ] infile [[OPTIONS... ] infile] ...
```

### Initialization

Flags:

- *-A* = Generate an AIFF output file.
- *-W* = Generate an WAV output file.
- *-h* = Generate an output file with no header.
- *-c* = Generate 8-bit signed\_char sound samples.
- *-a* = Generate alaw sound samples.
- *-u* = Generate ulaw sound samples.
- *-s* = Generate short integer sound samples.
- *-l* = Generate long (32 bit) integer sound samples.
- *-f* = Generate floating point samples.
- *-F arg* = Specifies the gain to be applied to the following input file. If arg is a floating point number that gain is applied uniformly to the input. Alternatively it could be a file name which specifies a breakpoint file for varying the gain for different periods.
- *-S integer* = Indicate at which sample to start to mix in the next input file.
- *-T fpnum* = Indicate at which time (in seconds) to start to mix in the next input file.
- *-1* = Mix in channel 1 from next sound file.
- *-2* = Mix in channel 2 from next sound file.
- *-3* = Mix in channel 3 from next sound file.
- *-4* = Mix in channel 4 from next sound file.
- *-^ intx inty* = Mix in channel x from next sound file as channel y in the output.
- *-v* = Verbose mode.
- *-R* = Continually rewrite the header while writing soundfile (WAV/AIFF).
- *-H integer* = Show a "heart-beat" to indicate progress, in style 1, 2 or 3.

- *-N* = Alert call (usually ringing the bell) when finished.
- *-o fname* = output file name (default: test.wav)

## Examples

The default values are

```
mixer -s -otest -F 1.0 -S 0
```

For example

```
mixer -F 0.96 in1.wav -S 300 -2 in2.aiff -S 300 -^4 1 in3.wav -o out.wav
```

This creates a new sound file with a constant gain of 0.96 from in1.wav with the second channel of in2.aiff mixed in after 300 samples and channel 4 of in3.wav output as channel 1 after 300 samples.

## Credits

Author: John ffitch

1994

## scale

scale — Scale the amplitude of a sound file.

### Description

Takes a sound file and scales it by applying a gain, either constant or variable. The scale can be specified as a multiplier, a maximum or a percentage of 0db.

### Syntax

```
scale [OPTIONS ... ] infile
```

### Initialization

Flags:

- *-A* = Generate an AIFF outout file.
- *-W* = Generate an WAV outout file.
- *-h* = Generate an outout file with no header.
- *-c* = Generate 8-bit signed\_char sound samples.
- *-a* = Generate alaw sound samples.
- *-u* = Generate ulaw sound samples.
- *-s* = Generate short integer sound samples.
- *-l* = Generate long (32 bit) integer sound samples.
- *-f* = Generate floating point samples.
- *-F arg* = Specifies the gain to be applied. If arg is a floating point number that gain is applied uniformly to the input. Alternatively it could be a file name which specifies a breakpoint file for varying the gain for different periods.
- *-M fnum* = Scales the input so the maximum absolute displacement is the value given.
- *-P fnum* = Scales the input so the maximum absolute displacement is the percentage given of 0db.
- *-R* = Continually rewrite the header while writing soundfile (WAV/AIFF).
- *-H integer* = Show a "heart-beat" to indicate progress, in style 1, 2 or 3.
- *-N* = Alert call (usually ringing the bell) when finished.
- *-o fname* = output file name (default: test.wav)

### Examples

```
scale -s -W -F 0.96 -o out.wav sound.wav
```

This creates a new sound file with a constant gain of 0.96. It is particularly useful if the input file is in floating point format.

## Credits

Author: John ffitich

1994

## mkdb

mkdb — Outputs a catalog of opcodes and plugin modules.

### Description

Outputs a catalog of opcodes and plugin modules, sorted by opcode name or by module.

### Syntax

```
mkdb [-m] [base_directory]
```

### Credits

Author: John ffitch

Aug 2011

New in version 5.14

## Credits

Dan Ellis

MIT Media Lab

Cambridge, Massachussetts

---

# Cscore

*Cscore* is an API (application programming interface) for generating and manipulating numeric score files. It is a part of the larger Csound API and includes a number of functions that can be called by a user-designed program written in the C language. *Cscore* can be invoked either as a standalone score preprocessor, or as part of a Csound performance by including the `-C` flag in its arguments:

```
cscore [scorefilein] [> scorefileout]
```

(where *cscore* is the name of your user-written program), or

```
csound [-C] [otherflags] [orchname] [scorename]
```

The available API functions augment the C language library functions; they can read either standard numeric scores or pre-sorted score files, can massage and expand the data in various ways, then make it available for performance by a Csound orchestra.

The user-written control program is written in C, and is compiled and linked to the Csound library (or the `csound` commandline program) by the user. It is not essential to know the C language well to write this program, since the function calls have a simple syntax, and are powerful enough to do most of the complicated work. Additional power can come from C later as the need arises.

The following sections explain all of the steps needed to make use of *Cscore*:

- *Events, Lists, and Operations* - Explains the syntax of *Cscore* functions and data structures.
- *Writing a Cscore Control Program* - Illustrates by example how to write your own control program.
- *Compiling a Cscore Program* - Outlines the steps for compiling and linking with the Csound library.
- *More Advanced Examples* - Addresses advanced issues such as multiple input scores and the details of running *Cscore* inside of a Csound performance.

## Events, Lists, and Operations

An event in *Cscore* is equivalent to one statement of a *standard numeric score* or a time-warped score (the format in which Csound writes a sorted score -- see any `score.srt`), and is stored internally in time-warped format. It is important to note that when *Cscore* is used in standalone-mode, it cannot understand any of the non-numeric "conveniences" that Csound allows in the input score format. Therefore, scores making use of features such as carry, ramp, expressions, and others will have to either be sorted first with the *scsort* utility or used with a modified *Csound* executable that contains the user's *Cscore* program. Score opcodes with macro arguments (r, m, n, and { }) are not understood.

Score events are each read in from an existing score file and stored in a C structure. The structures main components are an opcode and an array of pfield values. *Cscore* handles reading the events and storing them in memory for you. The format of the structure starts as follows:

```
typedef struct {
    CSHDR h;          /* space-managing header */
    char *strarg;     /* address of optional string argument */
    char op;         /* opcode-t, w, f, i, a, s or e */
    short pcnt;
    MYFLT p2orig;    /* unwarped p2, p3 */
    MYFLT p3orig;
```

```

    MYFLT p[1];      /* array of pfields p0, p1, p2 ... */
} EVENT;

```

MYFLT is either the C type *float* or *double* depending on how your copy of the Csound library was compiled. You should just declare any floating-point variables as MYFLT in your user program for compatibility.

Any *Cscore* function that creates, reads, or copies an event will return a pointer to the storage structure holding the event data. The event pointer can be used to access any component of the structure, in the form of *e->op* or *e->p[n]*. Each newly stored event will give rise to a new pointer, and a sequence of new events will generate a sequence of distinct pointers that must themselves be stored. Groups of event pointers are stored in an event list, which has its own structure:

```

typedef struct {
    CSHDR h;
    int nslots;      /* max events in this event list */
    int nevents;     /* number of events present */
    EVENT *e[1];    /* array of event pointers e0, e1, e2.. */
} EVLIST;

```

Any *Cscore* function that creates or modifies a list will return a pointer to the new list. The list pointer can be used to access any of its component event pointers, in the form of *a->e[n]*. Event pointers and list pointers are thus primary tools for manipulating the data of a score file. Pointers and lists of pointers can be copied and reordered without modifying the data values they refer to. This means that notes and phrases can be copied and manipulated from a high level of control. Alternatively, the data within an event or group of events can be modified without changing the event or list pointers. The *Cscore* API functions enable scores to be created and manipulated in this way.

With Csound 5, the names of all of the *Cscore* API functions have changed to be more explicit. In addition, each function now requires a pointer to a CSOUND object as its first argument. The structure of the CSOUND object is unimportant (and indeed cannot be modified in a user program). How to obtain this CSOUND pointer will be shown in the next section. The *Cscore* functions and data structures are available in the `cscore.h` header file, which you must include in your program code before you can use them.

The names of the *Cscore* functions specify whether they operate on single events or event lists. In the following summary of available function calls, some simple naming conventions are used:

```

The symbol cs is a pointer to a CSOUND object (CSOUND *);
The symbols e, f are pointers to events (notes);
The symbols a, b are pointers to lists (arrays) of such events;
The symbol n is an integer parameter of type int;
"..." indicates a string parameter (either a constant or variable of type char *);
The symbol fp is a score input stream file pointer (FILE *);

```

calling syntax	description
-----	-----
/* Functions for working with single events */	
e = cscoreCreateEvent(cs, n);	create a blank event with n pfields
e = cscoreDefineEvent(cs, "...");	defines an event as per the character string ...
e = cscoreCopyEvent(cs, f);	make a new copy of event f
e = cscoreGetEvent(cs);	read the next event in the score input file
cscorePutEvent(cs, e);	write event e to the score output file
cscorePutString(cs, "...");	write the string-defined event to score output
/* Functions for working with event lists */	
a = cscoreListCreate(cs, n);	create an empty event list with n slots
a = cscoreListAppendEvent(cs, a, e);	append event e to list a
a = cscoreListAppendStringEvent(cs, a, "...");	append a string-defined event to list a;
a = cscoreListCopy(cs, b);	copy the list b (but not the events)

```

a = cscoreListCopyEvents(cs, b);          copy the events of b, making a new list
a = cscoreListGetSection(cs);            read all events from score input, up to next s or e
a = cscoreListGetNext(cs, nbeats);      read next nbeats beats from score input (nbeats is MYFI
a = cscoreListGetUntil(cs, beatno);     read all events from score input up to beat beatno (MYFI
a = cscoreListSeparateF(cs, b);        separate the f statements from list b into list a
a = cscoreListSeparateTWF(cs, b);      separate the t,w & f statements from list b into list a
a = cscoreListAppendList(cs, a, b);    append the list b onto the list a
a = cscoreListConcatenate(cs, a, b);   concatenate (append) the list b onto the list a (same a
    cscoreListSort(cs, a);             sort the list a into chronological order by p[2]
n = cscoreListCount(cs, a);            returns the number of events in list a
a = cscoreListExtractInstruments(cs, b, "..."); extract notes of instruments ... (no new events)
a = cscoreListExtractTime(cs, b, from, to); extract notes of time-span, creating new events (from a
    cscoreListPut(cs, a);              write the events of list a to the score output file
    cscoreListPlay(cs, a);            send events of list a to the Csound orchestra for
                                        immediate performance (or print events if no orchestra)

/* Functions for reclaiming memory */
    cscoreFreeEvent(cs, e);           release the space of event e
    cscoreListFree(cs, a);            release the space of list a (but not the events)
    cscoreListFreeEvents(cs, a);     release the events of list a, and the list space

/* Functions for working with multiple input score files */
fp = cscoreFileGetCurrent(cs);         get the currently active scorefile pointer
                                        (initially finds the command-line input scorefile pointer)
fp = cscoreFileOpen(cs, "filename");   open another input scorefile (maximum of 5)
    cscoreFileSetCurrent(cs, fp);     make fp the currently active scorefile pointer
    cscoreFileClose(cs, fp);         close the scorefile relating to FILE *fp

```

Under Csound 4, the function names and parameters were as follows:

calling syntax	description
-----	-----
e = createv(n);	create a blank event with n pfields
e = defev("...");	defines an event as per the character string ...
e = copyev(f);	make a new copy of event f
e = getev();	read the next event in the score input file
putev(e);	write event e to the score output file
putstr("...");	write the string-defined event to score output
a = lcreat(n);	create an empty event list with n slots
int n;	
a = lappev(a,e);	append event e to list a
a = lappstrev(a,"...");	append a string-defined event to list a;
a = lcopy(b);	copy the list b (but not the events)
a = lcopyev(b);	copy the events of b, making a new list
a = lget();	read all events from score input, up to next s or e
a = lgetnext(nbeats);	read next nbeats beats from score input
float nbeats;	
a = lgetuntil(beatno);	read all events from score input up to beat beatno
float beatno;	
a = lsepf(b);	separate the f statements from list b into list a
a = lseptwf(b);	separate the t,w & f statements from list b into list a
a = lcat(a,b);	concatenate (append) the list b onto the list a
lsort(a);	sort the list a into chronological order by p[2]
a = lxins(b,"...");	extract notes of instruments ... (no new events)
a = lxtimev(b,from,to);	extract notes of time-span, creating new events
float from, to;	
lput(a);	write the events of list a to the score output file
lplay(a);	send events of list a to the Csound orchestra for immediate performance (or print events if no orchestra)
relev(e);	release the space of event e
lrel(a);	release the space of list a (but not the events)
lrelev(a);	release the events of list a, and the list space
fp = getcurfp();	get the currently active input scorefile pointer (initially finds the command-line input scorefile pointer)
fp = filopen("filename");	open another input scorefile (maximum of 5)
setcurfp(fp);	make fp the currently active scorefile pointer
filclose(fp);	close the scorefile relating to FILE *fp

# Writing a Cscore Control Program

The general format for a *Cscore* control program is:

```
#include "cscore.h"
void cscore(CSOUND *cs)
{
  /* VARIABLE DECLARATIONS */
  /* PROGRAM BODY */
}
```

The *include* statement will define the event and list structures and all of the *Cscore* API functions for the program. The name of the user function needs to be *cscore* if it will be linked with the standard main program in *cscoremain.c* or linked as the internal *Cscore* routine for a personal Csound executable. This *cscore()* function receives one argument from *cscoremain.c* or Csound -- *CSOUND \*cs* -- which is a pointer to a Csound object. The pointer *cs* must be passed as the first parameter to every *Cscore* API function that the program calls.

The following C program will read from a *standard numeric score*, up to (but not including) the first *s* or *e* statement, then write that data (unaltered) as output.

```
#include "cscore.h"
void cscore(CSOUND *cs)
{
  EVLIST *a; /* a is allowed to point to an event list */
  a = cscoreListGetSection(cs); /* read events in, return the list pointer */
  cscoreListPut(cs, a); /* write these events out (unchanged) */
  cscorePutString(cs, "e"); /* write the string e to output */
}
```

After execution of *cscoreListGetSection()*, the variable *a* points to a list of event addresses, each of which points to a stored event. We have used that same pointer to enable another list function -- *cscoreListPut()* -- to access and write out all of the events that were read. If we now define another symbol *e* to be an event pointer, then the statement

```
e = a->e[4];
```

will set it to the contents of the 4th slot in the *EVLIST* structure, *a*. The contents is a pointer to an event, which is itself comprised of an *array* of parameter field values. Thus the term *e->p[5]* will mean the value of parameter field 5 of the 4th event in the *EVLIST* denoted by *a*. The program below will multiply the value of that *pfield* by 2 before writing it out.

```
#include "cscore.h"
void cscore(CSOUND *cs)
{
  EVENT *e; /* a pointer to an event */
  EVLIST *a;
  a = cscoreListGetSection(cs); /* read a score as a list of events */
  e = a->e[4]; /* point to event 4 in event list a */
  e->p[5] *= 2; /* find pfield 5, multiply its value by 2 */
  cscoreListPut(cs, a); /* write out the list of events */
  cscorePutString(cs, "e"); /* add a "score end" statement */
}
```

Now consider the following score, in which *p[5]* contains frequency in Hz.

```
f 1 0 257 10 1
f 2 0 257 7 0 300 1 212 .8
i 1 1 3 0 440 10000
i 1 4 3 0 256 10000
i 1 7 3 0 880 10000
e
```

If this score were given to the preceding main program, the resulting output would look like this:

```
f 1 0 257 10 1
f 2 0 257 7 0 300 1 212 .8
i 1 1 3 0 440 10000
i 1 4 3 0 512 10000      ; p[5] has become 512 instead of 256.
i 1 7 3 0 880 10000
e
```

Note that the 4th event is in fact the second note of the score. So far we have not distinguished between notes and function table setup in a numeric score. Both can be classed as events. Also note that our 4th event has been stored in  $e[4]$  of the structure. For compatibility with Csound *pfield* notation, we will ignore  $p[0]$  and  $e[0]$  of the event and list structures, storing  $p1$  in  $p[1]$ , event 1 in  $e[1]$ , etc. The *Cscore* functions all adopt this convention.

As an extension to the above, we could decide to use the same pointers  $a$  and  $e$  to examine each of the events in the list. Note that  $e$  was not set to the numeral 4, but to the location of the 4th slot in the list. To inspect  $p5$  of the previous event in the list, we need only redefine  $e$  with the assignment

```
e = a->e[3];
```

and reference the 5th slot of the *pfield* array using the expression

```
e->p[5]
```

More generally, we can use an integer variable as an index to the array  $e[]$ , and access each event in sequence by using a loop and incrementing the index. The number of events stored in an *EVLIST* is contained in the *nevents* member of the struct.

```
int index;      /* start with e[1] because e[0] is not used */
for (index = 1; index <= a->nevents; index++)
{
    e = a->e[index];
    /* do something with e */
}
```

The above example starts with  $e[1]$  and increases the index each time through the loop ( $index++$ ) until it is greater than  $a->nevents$ , the index of the last event in the list. The statements inside the *for* loop do execute a final time when  $index$  equals  $a->nevents$ .

In the following program we will use the same input score. This time we will separate the *ftable* statements from the *note* statements. We will next write the three note-events stored in the list  $a$  to the output, then create a second score section consisting of the original pitch set and a transposed version of itself. This will bring about an octave doubling.

Here, our index to the array is  $n$  and we increment  $n$  as part of a *for* block which iterates *nevents* times, allowing one statement to act upon the same *pfield* of each successive event.

```
#include "cscore.h"
void cscore(CSOUND *cs)
{
```

```

EVENT *e, *f;
EVLIST *a, *b;
int n;

a = cscoreListGetSection(cs);          /* read score into event list "a" */
b = cscoreListSeparateF(cs, a);        /* separate f statements */
cscoreListPut(cs, b);                  /* write f statements out to score */
cscoreListFreeEvents(cs, b);           /* and release the spaces used */
e = cscoreDefineEvent(cs, "t 0 120");  /* define event for tempo statement */
cscorePutEvent(cs, e);                 /* write tempo statement to score */
cscoreListPut(cs, a);                  /* write the notes */
cscorePutString(cs, "s");              /* section end */
cscorePutEvent(cs, e);                 /* write tempo statement again */
b = cscoreListCopyEvents(cs, a);       /* make a copy of the notes in "a" */
for (n = 1; n <= b->nevents; n++)      /* iterate the following lines nevents times: */
{
    f = b->e[n];
    f->p[5] *= 0.5;                     /* transpose pitch down one octave */
}
a = cscoreListAppendList(cs, a, b);    /* now add these notes to original pitches */
cscoreListPut(cs, a);
cscorePutString(cs, "e");
}

```

The output of this program is:

```

f 1 0 257 10 1
f 2 0 257 7 0 300 1 212 .8
t 0 120
i 1 1 3 0 440 10000
i 1 4 3 0 256 10000
i 1 7 3 0 880 10000
s
t 0 120
i 1 1 3 0 440 10000
i 1 4 3 0 256 10000
i 1 7 3 0 880 10000
i 1 1 3 0 220 10000
i 1 4 3 0 128 10000
i 1 7 3 0 440 10000
e

```

If the output is only being written to a file, then the unsorted order of the events is not a problem. The output is written to a file (or standard output) whenever the function *cscoreListPut()* is used. However, if this program were to be called during a Csound performance and the function *cscoreListPlay()* replaced *cscoreListPut()*, then the events would be sent to the orchestra instead of to a file and they should then be sorted beforehand by calling the function *cscoreListSort()*. The details of score output and playing when using *Cscore* from within Csound are described in the next section.

Next we extend the above program by using the *for* loop to look at *p[5]* and *p[6]*. In the original score *p[6]* denotes amplitude. To create a *diminuendo* in the added lower octave, which is independent from the original set of notes, a variable called *dim* will be used.

```

#include "cscore.h"
void cscore(CSOUND *cs)
{
    EVENT *e, *f;
    EVLIST *a, *b;
    int n, dim;                          /* declare two integer variables */

    a = cscoreListGetSection(cs);
    b = cscoreListSeparateF(cs, a);
    cscoreListPut(cs, b);

```

```

cscoreListFreeEvents(cs, b);
e = cscoreDefineEvent(cs, "t 0 120");
cscorePutEvent(cs, e);
cscoreListPut(cs, a);
cscorePutString(cs, "s");
cscorePutEvent(cs, e);           /* write out another tempo statement */
b = cscoreListCopyEvents(cs, a);
dim = 0;                          /* initialize dim to 0 */
for (n = 1; n <= b->nevents; n++)
{
    f = b->e[n];
    f->p[6] -= dim;                /* subtract current value of dim */
    f->p[5] *= 0.5;                /* transpose pitch down one octave */
    dim += 2000;                  /* increase dim for each note */
}
a = cscoreListAppendList(cs, a, b); /* now add these notes to original pitches */
cscoreListPut(cs, a);
cscorePutString(cs, "e");
}

```

Using the same input score again, the output from this program is:

```

f 1 0 257 10 1
f 2 0 257 7 0 300 1 212 .8
t 0 120
i 1 1 3 0 440 10000
i 1 4 3 0 256 10000
i 1 7 3 0 880 10000
s
t 0 120
i 1 1 3 0 440 10000      ; Three original notes at
i 1 4 3 0 256 10000    ; beats 1,4 and 7 with no dim.
i 1 7 3 0 880 10000
i 1 1 3 0 220 10000    ; three notes transposed down one octave
i 1 4 3 0 128 8000     ; also at beats 1,4 and 7 with dim.
i 1 7 3 0 440 6000
e

```

In the following program the same three-note sequence will be repeated at various time intervals. The starting time of each group is determined by the values of the array *cue*. This time the *dim* will occur for each group of notes rather than each note. Note the position of the statement which increments the variable *dim* outside the inner *for* loop.

```

#include "cscore.h"
int cue[3] = {0,10,17};           /* declare an array of 3 integers */
void cscore(CSOUND *cs)
{
    EVENT *e, *f;
    EVLIST *a, *b;
    int n, dim, cuecount;         /* declare new variable cuecount */

    a = cscoreListGetSection(cs);
    b = cscoreListSeparateF(cs, a);
    cscoreListPut(cs, b);
    cscoreListFreeEvents(cs, b);
    e = cscoreDefineEvent(cs, "t 0 120");
    cscorePutEvent(cs, e);
    dim = 0;
    for (cuecount = 0; cuecount <= 2; cuecount++) /* elements of cue are numbered 0, 1, 2 */
    {
        for (n = 1; n <= a->nevents; n++)
        {
            f = a->e[n];
            f->p[6] -= dim;

```

```

        f->p[2] += cue[cuecount];          /* add values of cue */
    }
    printf("; diagnostic: cue = %d\n", cue[cuecount]);
    dim += 2000;
    cscoreListPut(cs, a);
}
cscorePutString(cs, "e");
}

```

Here the inner *for* loop looks at the events of list *a* (the notes) and the outer *for* loop looks at each *repetition* of the events of list *a* (the pitch group "cues"). This program also demonstrates a useful trouble-shooting device with the *printf* function. The *semi-colon* is first in the character string to produce a comment statement in the resulting score file. In this case the value of *cue* is being printed in the output to insure that the program is taking the proper *array* member at the proper time. When output data is wrong or error messages are encountered, the *printf* function can help to pinpoint the problem.

Using the same input file, the C program above will generate the following score. Can you determine why the last set of notes starts at the wrong time and how to correct the problem?

```

f 1 0 257 10 1
f 2 0 257 7 0 300 1 212 .8
t 0 120
; diagnostic: cue = 0
i 1 1 3 0 440 10000
i 1 4 3 0 256 10000
i 1 7 3 0 880 10000
; diagnostic: cue = 10
i 1 11 3 0 440 8000
i 1 14 3 0 256 8000
i 1 17 3 0 880 8000
; diagnostic: cue = 17
i 1 28 3 0 440 4000
i 1 31 3 0 256 4000
i 1 34 3 0 880 4000
e

```

## Compiling a Cscore Program

A *Cscore* program can be invoked either as a *standalone program* or as part of *Csound* in between sorting the score and performing the score with the orchestra:

```
cscore [scorefilein] [> scorefileout]
```

or

```
csound [-C] [otherflags] [orchname] [scorename]
```

Before trying to compile your own *Cscore* program, you will most likely want to obtain a copy of the *Csound* source code. Either download the latest source distribution for your platform or check out a copy of the *csound5* module from Sourceforge CVS. There are several files in the sources that will help you. Within the `examples/cscore/` directory are a number of examples of *Cscore* control programs, including all of the examples contained in this manual. And in the `frontends/cscore/` directory are the two files `cscoremain.c` and `cscore.c`. `cscoremain.c` contains a simple *main* function that performs all of the initialization that a standalone *Cscore* program needs to do before it calls your control function. This main “stub” initializes *Csound*, reads the commandline arguments, opens the input and output score files, and then calls a function `cscore()`. As described above, it is expected that you will write the `cscore()` function and provide it in another file. The file `frontends/cscore/cscore.c` shows the simplest example of a `cscore()` function that reads in a score of any length and writes it to the output unchanged.

So, to create a standalone program, write a control program as shown in the previous section. Let's assume that you saved this program in a file named "myscore.c". Next, you need to compile and link this program with the Csound library and *cscoremain.c* in order to create an executable by following the set of directions below that apply to your operating system. It will be helpful to already have some familiarity with the C compiler on your computer since the information below cannot be complete for all possible systems.

## Linux and Unix

The following commands assume that you have copied your file *myscore.c* into the same directory as *cscoremain.c*, that you have opened a terminal to that same directory, and that you have previously installed a binary distribution of Csound that placed a library *libcsound.a* or *libcsound.so* into */usr/local/lib* and the header files for the Csound API into */usr/local/include/csound*.

To compile and link:

```
gcc myscore.c cscoremain.c -o cscore -lcsound -L/usr/local/lib -I/usr/local/include/csound
```

To run (sending the results to standard output):

```
./cscore test.sco
```

It is possible that on some Unix systems, the C compiler will be named *cc* or something else other than *gcc*.

## Windows

Csound is usually compiled on Windows using the MinGW environment that makes GCC -- the same compiler used on Linux -- available using a Unix-like command shell (MSYS). Since pre-compiled libraries for Csound on Windows are built in this way, you may need to use MinGW as well to link to them. If you have built Csound using another compiler, then you should be able to build *Cscore* with that compiler as well.

Compiling standalone *Cscore* programs using MinGW should be similar to the procedure for Linux above with library and header paths changed appropriately for where Csound is installed on the Windows system. *(Please feel free to contribute more detailed instructions here as the editor has been unable to test Cscore on a Windows machine).*

## OS X

The following commands assume that you have copied your file *myscore.c* into the same directory as *cscoremain.c* and that you have opened a terminal to that same directory. In addition, the Apple-supplied developer tools (including the GCC compiler) should be installed on your system and you should have previously installed a binary distribution of Csound that placed the CsoundLib framework into */Library/Frameworks*.

Use this command compile and link. (You may get a warning about "multiple definitions of symbol *\_cscore*").

```
gcc cscore.c cscoremain.c -o cscore -framework CsoundLib -I/Library/Frameworks/CsoundLib.framework/Head
```

To run (sending the results to standard output):

```
./cscore test.sco
```

## MacOS 9

You will need CodeWarrior or some other development environment installed on your computer (MPW may work). Download the source code distribution for OS 9 (it will have a name like *Csound5.05\_OS9\_src.smi.bin*).

If using CodeWarrior, find and open the project file "Cscore5.cw8.mcp" in the folder "Csound5.04-OS9-source:macintosh:Csound5Library:". This project file is configured to use the source files *cscore.c* and *cscoremain\_MacOS9.c* from the csound5 source tree and the Csound5Lib shared library produced by compiling Csound with the "Csound5.cw8.mcp" project file. You should substitute your own *Cscore* program file for *cscore.c* and either compile Csound5Lib first or substitute a copy of the library in the project from the binary distribution of Csound for OS 9. The file *cscoremain\_MacOS9.c* contains specialized code for configuring CodeWarrior's SIOUX console library and allows commandline arguments to be entered before the program is run.

Once you have the proper files included in the project window, click the "Make" button and CodeWarrior should produce an application named "*Cscore*". When you run this application, it first displays a window allowing you to type in the arguments to the main function. You only need to type in the filename or pathname to the input score -- do not type in "cscore". The input file should be in the same folder as the application or else you will need to type a full or relative pathname to the file. Output will be displayed in the console window. You can use the *Save* command from the *File* menu before quitting if you wish. Alternatively, in the commandline dialog, you can choose to redirect the output to a file by clicking on the *File* button on the right side of the dialog. (Note that the console window can only display about 32,000 characters, so writing to a file is necessary for long scores).

## Making Cscore usable from within Csound

To operate from Csound, first follow the instructions for compiling Csound (see *Building Csound*) according to the operating system that you are using. Once you have successfully built an unmodified Csound system, then substitute your own *cscore()* function for the one in the file *Top/cscore\_internal.c*, and rebuild Csound.

The resulting executable is your own special Csound, usable as above. The *-C flag* will invoke your *Cscore* program after the input score is sorted into "*score.srt*". The details of what happens when you run Csound with the *-C* flag are given in the next section.

Csound 5 also provides an additional way to run your own *Cscore* program from within Csound. Using the API, a host application can set a *Cscore callback function*, which is a function that Csound will call instead of using the built-in *cscore()* function. One advantage of this approach is that it is not necessary to recompile the entirety of Csound. Another benefit is that the host application can select at runtime from more than one *Cscore* function to designate as the callback. The disadvantage is that you need to write a host application.

A simple approach to using a *Cscore* callback via the API would be to modify the standard Csound main program -- which is a simple Csound host -- contained in the file *frontends/csound/csound\_main.c*. Adding a call to *csoundSetCscoreCallback()* after the call to *csoundCreate()* but before the call to *csoundCompile()* should do the job. Recompiling this file and linking to an existing Csound library will make a commandline version of Csound that works similarly to the one described above. Don't forget to use the *-C* flag.

## Notes about score formats and run-time behavior

As stated previously, the input files to *Cscore* may be in original or time-warped and pre-sorted form; this modality will be preserved (section by section) in reading, processing, and writing scores. Standalone processing will most often use unwarped sources and create unwarped new files. When running from within

Csound, the input score will arrive already warped and sorted, and can thus be sent directly (normally section by section) to the orchestra. One advantage of this method of using *Cscore* is that all of the syntactical conveniences of the full Csound score language may be used -- macros, arithmetic expressions, carry, ramp, etc. -- since the score will go through the "Carry, Tempo, Sort" phases of score processing before being passed to the user-supplied *Cscore* program.

When running within Csound, a list of events can be conveyed to a Csound orchestra using *cscoreListPlay()*. There may be any number of *cscoreListPlay()* calls in a *Cscore* program. Each list so conveyed can be either time-warped or not, but each list must be in strict *p2*-chronological order (either from presorting or using *cscoreListSort()*). If there is no *cscoreListPlay()* in a *Cscore* module run from within Csound, all events written out (via *cscorePutEvent()*, *cscorePutString()*, or *cscoreListPut()*) are written to a new score in the current directory with the name "*cscore.out*". Csound then invokes the score sorter again before sending this new score to the orchestra for performance. The final, sorted, output score is written to a file named "*cscore.srt*".

A standalone *Cscore* program will normally use the "put" commands to write into its output file. If a standalone *Cscore* program calls *cscoreListPlay()*, the events thus intended for performance will be sent to the output in the same way as if *cscoreListPut()* had been called instead.

A note list sent by *cscoreListPlay()* for performance should be temporally distinct from subsequent note lists. No note-end should extend past the next list's start time, since *cscoreListPlay()* will complete each list before starting the next (i.e. like a Section marker that doesn't reset local time to zero). This is important when using *cscoreListGetNext()* or *cscoreListGetUntil()* to fetch and process score segments prior to performance, because these functions may only read part of an unsorted section.

## More Advanced Examples

The following program demonstrates reading from two different input files. The idea is to switch between two 2-section scores, and write out the interleaved sections to a single output file.

```
#include "cscore.h"          /* CSCORE_SWITCH.C */
cscore(CSOUND* cs)         /* callable from either Csound or standalone cscore */
{
    EVLIST *a, *b;
    FILE *fp1, *fp2;          /* declare two scorefile stream pointers */
    fp1 = cscoreFileGetCurrent(cs); /* this is the command-line score */
    fp2 = cscoreFileOpen(cs, "score2.srt"); /* this is an additional score file */
    a = cscoreListGetSection(cs); /* read section from score 1 */
    cscoreListPut(cs, a); /* write it out as is */
    cscorePutString(cs, "s");
    cscoreFileSetCurrent(cs, fp2);
    b = cscoreListGetSection(cs); /* read section from score 2 */
    cscoreListPut(cs, b); /* write it out as is */
    cscorePutString(cs, "s");
    cscoreListFreeEvents(cs, a); /* optional to reclaim space */
    cscoreListFreeEvents(cs, b);
    cscoreFileSetCurrent(cs, fp1);
    a = cscoreListGetSection(cs); /* read next section from score 1 */
    cscoreListPut(cs, a); /* write it out */
    cscorePutString(cs, "s");
    cscoreFileSetCurrent(cs, fp2);
    b = cscoreListGetSection(cs); /* read next sect from score 2 */
    cscoreListPut(cs, b); /* write it out */
    cscorePutString(cs, "e");
}

```

Finally, we show how to take a literal, uninterpreted score file and imbue it with some expressive timing changes. The theory of composer-related metric pulses has been investigated at length by Manfred Clynes,

and the following is in the spirit of his work. The strategy here is to first create an *array* of new *onset* times for every possible sixteenth-note onset, then to index into it so as to adjust the start and duration of each note of the input score to the interpreted time-points. This also shows how a Csound orchestra can be invoked repeatedly from a run-time score generator.

```
#include "cscore.h"          /*  CSCORE_PULSE.C  */

/* program to apply interpretive durational pulse to      */
/* an existing score in 3/4 time, first beats on 0, 3, 6 ... */

static float four[4] = { 1.05, 0.97, 1.03, 0.95 }; /* pulse width for 4's */
static float three[3] = { 1.03, 1.05, .92 }; /* pulse width for 3's */

cscore(CSOUND* cs)          /* This example should be called from Csound */
{
    EVLIST *a, *b;
    EVENT *e, **ep;
    float pulse16[4*4*4*4*3*4]; /* 16th-note array, 3/4 time, 256 measures */
    float acc16, acc1, incl, acc3, inc3, acc12, incl2, acc48, inc48, acc192, incl92;
    float *p = pulse16;
    int n16, n1, n3, n12, n48, n192;

    /* fill the array with interpreted ontimes */
    for (acc192=0., n192=0; n192<4; acc192+=192.*inc192, n192++)
        for (acc48=acc192, incl92=four[n192], n48=0; n48<4; acc48+=48.*inc48, n48++)
            for (acc12=acc48, inc48=inc192*four[n48], n12=0; n12<4; acc12+=12.*inc12, n12++)
                for (acc3=acc12, incl2=inc48*four[n12], n3=0; n3<4; acc3+=3.*inc3, n3++)
                    for (acc1=acc3, inc3=inc12*four[n3], n1=0; n1<3; acc1+=incl, n1++)
                        for (acc16=acc1, incl=inc3*three[n1], n16=0; n16<4; acc16+=.25*inc1*four[n16], n16++)
                            *p++ = acc16;

    /* for (p = pulse16, n1 = 48; n1--; p += 4) /* show vals & diffs */
    /* printf("%g %g %g %g %g %g %g %g\n", *p, *(p+1), *(p+2), *(p+3),
    /* *(p+1)-*p, *(p+2)-*(p+1), *(p+3)-*(p+2), *(p+4)-*(p+3)); */

    a = cscoreListGetSection(cs); /* read sect from tempo-warped score */
    b = cscoreListSeparateTWF(cs, a); /* separate warp & fn statements */
    cscoreListPlay(cs, b); /* and send these to performance */
    a = cscoreListAppendStringEvent(cs, a, "s"); /* append a sect statement to note list */
    cscoreListPlay(cs, a); /* play the note-list without interpretation */
    for (ep = &a->e[1], n1 = a->nevents; n1--; ) { /* now pulse-modify it */
        e = *ep++;
        if (e->op == 'i') {
            e->p[2] = pulse16[(int)(4. * e->p2orig)];
            e->p[3] = pulse16[(int)(4. * (e->p2orig + e->p3orig))] - e->p[2];
        }
    }

    cscoreListPlay(cs, a); /* now play modified list */
}

```

---

# Csbeats

*Csbeats* is an alternative score language that is aimed at specifying simple scores in standard western tunings and rhythms. *Csbeats* can be invoked via the CsScore component of a standard .csd score with *bin="csbeats"* or as stand-alone program which generates a standard numeric score.

As a stand-alone the program reads standard input and writes to standard output.

The *csbeats* language is very simple, having only 5 kinds of statement, and only one of them has any complexity. In general the introductory word for each statement type is case insensitive, so "QUIT", "quit", "Quit"... are all the same. Comments can be introduced in either ANSI C89 format or C++ (that is either /\* ... \*/ or // to the end of line) or Csound's semicolon.

- *QUIT*

Causes csbeats to exit. For flexibility the command *END* is also accepted for the identical action.

- *BEATS=integer*

Sets the number of beats per minute for the following score until the end or until it is reset. The default value is 60bpm. The token *BPS* is also acceptable instead of BEATS.

- *PERMEASURE=integer*

Sets the number of beats in a bar. The default value is 4.

- *BAR*

Start a new bar.

- *BAR integer*

Start the bar whose number is given.

- *i integer attributes*

Specified a note event for the numbered instrument. The attributes may be any of a pitch, duration or dynamic, or a positioning of the note to a beat or measure, and can be in any order.

Pitches are specified with a conventional note name (English form) in upper case optionally followed by a #, x (for double sharp), b (for flat) or bb (for double flat). A note of Z is a rest (think zzzz). All notes except rests must be followed by an octave number, with A4 being international A (440Hz). Pitches are passed to Csound in Hertz in the parameter p4, and are twelve tone equal temperament.

Durations are coded in lower case with the initial letter of the name for whole, half, quarter, eighth and sixteenth notes, or *th* for thirtysecondth notes. Except for *w* these can be moderated by appending as suffix:

- *d* or *.* Dotted note (half as long again)
- *dd* or *..* Dotted note (three quarters as long again)
- *t* Triplet note (third notes in two)
- *t.* Dotted triplet note

- *q* Quintuplet (Five notes in four)

- *s* Septuplet (Seven notes in eight)

Durations can be added together by giving more than one duration. To make this more intuitive a + sign can be used instead of white space.

The dynamics are written in conventional notation, that is fff, ff, f, mf, mp, p, pp, ppp, pppp. These are passed to the instrument as p5 as 0 for fff, and one less dB for each step below. The default dynamic is fortissimo.

Additional parameters (p-fields) can be specified in an instrument statement by giving statements like *p6=42* which would set the p6 field to 42 until it is changed. The number must be 6 or greater as the first 5 fields are taken with fixed information. Each instrument number has its own additional parameters.

If any of these attributes is missing it carried forward from the previous note, with beat position being incremented to the end of the previous note.

In addition an event can be directed to a particular measure with an m attribute or a particular beat with a b.

The opening of Bach's Goldberg variation number 3 can be coded as:

```
; Bach - Goldberg Variations - Variato 3
; by Brian Baughn 3-14-05
; bbaughn@berklee.net
beats = 120
permeasure = 6

i101    m1 b1 B4 mp qd+s
i101          C5    s
i101          D5
i101          C5
i101          D5
i101          E5
i101          A4    qd+s
i101          B4    s
i101          C5
i101          B4
i101          C5
i101          D5

i101    m2 b1 G4    qd
i101          G5    qd+e
i101          A5    s
i101          G5
i101          F#5
i101          G5
i101          A5    e

i101    m3 b1.5 D5    s
i101          C5
i101          B4
i101          A4
i101          B4    e
i101          C5    s
i101          B4
i101          A4
i101          B4
i101          G4    e
i101          E5
i101          D5
```

```

i101      C5
i101      F#5
i101      A5

i101  m4 b1  B4  q
i101      G5  e
i101      G5  q
i101      F#5 e
i101      Z   e // Z is a rest (zzzzz..)
i101      e
i101      B5  e
i101      A5  q
i101      D5  e

quit

```

This produces output

```

;;;setting bpm=120.000000
;;;setting permeasure=6
i101 0.000000 0.875000 493.883621 -4
i101 0.875000 0.125000 523.251131 -4
i101 1.000000 0.125000 587.329536 -4
i101 1.125000 0.125000 523.251131 -4
i101 1.250000 0.125000 587.329536 -4
i101 1.375000 0.125000 659.255114 -4
i101 1.500000 0.875000 440.000000 -4
i101 2.375000 0.125000 493.883621 -4
i101 2.500000 0.125000 523.251131 -4
i101 2.625000 0.125000 493.883621 -4
i101 2.750000 0.125000 523.251131 -4
i101 2.875000 0.125000 587.329536 -4
i101 3.000000 0.750000 391.995436 -4
i101 3.750000 1.000000 783.990872 -4
i101 4.750000 0.125000 880.000000 -4
i101 4.875000 0.125000 783.990872 -4
i101 5.000000 0.125000 739.988845 -4
i101 5.125000 0.125000 783.990872 -4
i101 5.250000 0.250000 880.000000 -4
i101 6.250000 0.125000 587.329536 -4
i101 6.375000 0.125000 523.251131 -4
i101 6.500000 0.125000 493.883621 -4
i101 6.625000 0.125000 440.000000 -4
i101 6.750000 0.250000 493.883621 -4
i101 7.000000 0.125000 523.251131 -4
i101 7.125000 0.125000 493.883621 -4
i101 7.250000 0.125000 440.000000 -4
i101 7.375000 0.125000 493.883621 -4
i101 7.500000 0.250000 391.995436 -4
i101 7.750000 0.250000 659.255114 -4
i101 8.000000 0.250000 587.329536 -4
i101 8.250000 0.250000 523.251131 -4
i101 8.500000 0.250000 739.988845 -4
i101 8.750000 0.250000 880.000000 -4
i101 9.000000 0.500000 493.883621 -4
i101 9.500000 0.250000 783.990872 -4
i101 9.750000 0.500000 783.990872 -4
i101 10.250000 0.250000 739.988845 -4
;;;rest at 10.500000 for 0.250000
;;;rest at 10.750000 for 0.250000
i101 11.000000 0.250000 987.767243 -4
i101 11.250000 0.500000 880.000000 -4
i101 11.750000 0.250000 587.329536 -4
e

```

## A Complete Example

Here is a simple example of the *csbeats* score generator. It uses the file *csbeats.csd* [examples/csbeats.csd].

### Example 1233. A simple example of csbeats.

See the sections *Real-time Audio* and *Command Line Flags* for more information on using command line flags.

```

<CsoundSynthesizer>
<CsOptions>
</CsOptions>

<CsInstruments>
sr      =          44100
nchnls  =    2

gi1 ftgen 1, 0, 4096, 10, 1
gi2 ftgen 2, 0, 4096, 7, -1, 4096, 1 ; sawtooth
gi3 ftgen 3, 0, 4096, 7, 0, 1024, 1, 2048, -1, 1024, 0 ;triangle

instr 101,102,103
  iamp =      ampdbfs(p5)
  a1  oscil  iamp, p4, p1-100
  kenv expseg 1, p3, .01
  a1  =      a1 * kenv
      outs  a1, a1
endin

</CsInstruments>
<CsScore bin="csbeats">
; by Brian Baughn 3-14-05
; bbaughn@berklee.net

beats = 100
permeasure = 4

i101  m1 b1  q  mp  D3
i101  F3
i101  D3

i101  m2 b1  D3
i101  b3  D3

i101  m3 b1  D3
i101  F3
i101  D3

i101  m4 b1  D3
i101  b3  D3

i101  m5 b1  D3
i101  b4  G5

i101  m6 b1  E5
i101  b2  F5
i101  b3  e  Eb5
i101  b3.5 e
i101  b4  q

i101  m7 b1  e  D5
i101

```

```
i101          q
i101          e      Db5
i101
i101          q

i101  m8 b1  q      D5
i101          E5
i101          D5

i102  m1 b2  q      D4
i102    b4          E4
i102    b4          Bb3

i102  m2 b2          F4
i102    b2          B3
i102    b4          C#4
i102    b4          Bb3

i102  m3 b2  q      D4
i102    b4          E4
i102    b4          Bb3

i102  m4 b2          F4
i102    b2          B3
i102    b4          C#4
i102    b4          Bb3

i103  m5 b2  e      F6
i103    b2    e      A5
i103    b2.5  e      D6
i103    b3    e      F6
i103    b3    e      A5
i103    b4    e      E6

i103  m6 b1  q      C#6
i103          q      D6
i103          e      C6
i103
i103          q

i103  m7 b1  e      B5
i103
i103          q
i103    e      Bb5
i103
i103          q

i103  m8 b1  e      F5
i103    b1    e      A5
i103    b1.5  e      D6
i103    b2    e      Bb5
i103    b2.5  e      D6
i103    b3    q      F5
i103    b3    A5
end
</CsScore>

</CsoundSynthesizer>
```

---

# **Part IV. Opcode Quick Reference**

---

---

# Table of Contents

Opcode Quick Reference .....	3400
------------------------------	------

---

# Opcode Quick Reference

## Orchestra Syntax:Header.

```
Odbfs = iarg
Odbfs

A4 = iarg

kr = iarg

ksmps = iarg

nchnls = iarg

nchnls_i = iarg

sr = iarg
```

## Orchestra Syntax:Block Statements.

```
endin

endop

instr i, j, ...

opcode name, outtypes, intypes
```

## Orchestra Syntax:Macros.

```
#define NAME # replacement text #

#define NAME(a' b' c') # replacement text #

$NAME

#ifdef NAME
....
#else
....
#endif

#ifndef NAME
....
#else
....
#endif

#include "filename"

#undef NAME
```

## Signal Generators:Additive Synthesis/Resynthesis.

```
ares adsyn kamod, kfmod, ksmod, ifilcod
ares adsynt kamp, kcps, iwfn, ifreqfn, iampfn, icnt [, iphs]
ar adsynt2 kamp, kcps, iwfn, ifreqfn, iampfn, icnt [, iphs]
ares hsboscil kamp, ktone, kbrite, ibasfreq, iwfn, ioctfn \
    [, ioctcnt] [, iphs]
```

### Signal Generators:Basic Oscillators.

```
kres lfo kamp, kcps [, itype]
ares lfo kamp, kcps [, itype]

ares oscbnk kcps, kamd, kfmd, kpmd, iovrlap, iseed, kllminf, kllmaxf, \
    kl2minf, kl2maxf, ilfomode, keqminf, keqmaxf, keqminl, keqmaxl, \
    keqminq, keqmaxq, ieqmode, kfn [, illfn] [, il2fn] [, ieqffn] \
    [, ieqlfn] [, ieqqfn] [, itabl] [, ioutfn]

ares oscil xamp, xcps [, ifn, iphs]
kres oscil kamp, kcps [, ifn, iphs]

ares oscil3 xamp, xcps [, ifn, iphs]
kres oscil3 kamp, kcps [, ifn, iphs]

ares oscili xamp, xcps[, ifn, iphs]
kres oscili kamp, kcps[, ifn, iphs]

ares oscilikt xamp, xcps, kfn [, iphs] [, istor]
kres oscilikt kamp, kcps, kfn [, iphs] [, istor]

ares osciliktp kcps, kfn, kphs [, istor]

ares oscilikts xamp, xcps, kfn, async, kphs [, istor]

ares osciln kamp, ifrq, ifn, itimes

ares oscils iamp, icps, iphs [, iflg]

ares poscil aamp, acps [, ifn, iphs]
ares poscil aamp, kcps [, ifn, iphs]
ares poscil kamp, acps [, ifn, iphs]
ares poscil kamp, kcps [, ifn, iphs]
ires poscil kamp, kcps [, ifn, iphs]
kres poscil kamp, kcps [, ifn, iphs]

ares poscil3 aamp, acps [, ifn, iphs]
ares poscil3 aamp, kcps [, ifn, iphs]
ares poscil3 kamp, acps [, ifn, iphs]
ares poscil3 kamp, kcps [, ifn, iphs]
ires poscil3 kamp, kcps [, ifn, iphs]
kres poscil3 kamp, kcps [, ifn, iphs]

kout vibr kAverageAmp, kAverageFreq, ifn

kout vibrato kAverageAmp, kAverageFreq, kRandAmountAmp, kRandAmountFreq, kAmpMinRate, \
    kAmpMaxRate, kcpsMinRate, kcpsMaxRate, ifn [, iphs]
```

### Signal Generators:Dynamic Spectrum Oscillators.

```
ares buzz xamp, xcps, knh, ifn [, iphs]

ares gbuzz xamp, xcps, knh, klh, kmul, ifn [, iphs]

ares mpulse kamp, kintvl [, ioffset]

ares vco xamp, xcps, iwave, kpw [, ifn] [, imaxd] [, ileak] [, inyx] \
    [, iphs] [, iskip]

ares vco2 kamp, kcps [, imode] [, kpw] [, kphs] [, inyx]

kfn vco2ft kcps, iwave [, inyx]

ifn vco2ift icps, iwave [, inyx]

ifn vco2init iwave [, ibasfn] [, ipmul] [, iminsiz] [, imaxsiz] [, isrcft]
```

### Signal Generators:FM Synthesis.

```
a1, a2 crossfm xfrq1, xfrq2, xndx1, xndx2, kcps, ifn1, ifn2 [, iphs1] [, iphs2]
a1, a2 crossfmi xfrq1, xfrq2, xndx1, xndx2, kcps, ifn1, ifn2 [, iphs1] [, iphs2]
a1, a2 crosspm xfrq1, xfrq2, xndx1, xndx2, kcps, ifn1, ifn2 [, iphs1] [, iphs2]
a1, a2 crosspmi xfrq1, xfrq2, xndx1, xndx2, kcps, ifn1, ifn2 [, iphs1] [, iphs2]
a1, a2 crossfmpm xfrq1, xfrq2, xndx1, xndx2, kcps, ifn1, ifn2 [, iphs1] [, iphs2]
a1, a2 crossfmpmi xfrq1, xfrq2, xndx1, xndx2, kcps, ifn1, ifn2 [, iphs1] [, iphs2]

ares fmb3 kamp, kfreq, kc1, kc2, kvdepth, kvrate[, ifn1, ifn2, ifn3, \
    ifn4, ivfn]

ares fmbell kamp, kfreq, kc1, kc2, kvdepth, kvrate[, ifn1, ifn2, ifn3, \
    ifn4, ivfn, isus]

ares fmmetal kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, \
    ifn4, ivfn

ares fmpercfl kamp, kfreq, kc1, kc2, kvdepth, kvrate[, ifn1, ifn2, \
    ifn3, ifn4, ivfn]

ares fmrhode kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, \
    ifn3, ifn4, ivfn

ares fmvoice kamp, kfreq, kvowel, ktilt, kvibamt, kvibrate[, ifn1, \
    ifn2, ifn3, ifn4, ivibfn]

ares fmwurlie kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, \
    ifn4, ivfn

ares foscil xamp, kcps, xcar, xmod, kndx, ifn [, iphs]

ares foscili xamp, kcps, xcar, xmod, kndx, ifn [, iphs]
```

### Signal Generators:Granular Synthesis.

```
asig diskgrain Sfname, kamp, kfreq, kpitch, kgrsize, kprate, \
    ifun, iolaps [,imaxgrsize , ioffset]

ares fof xamp, xfund, xform, koct, kband, kris, kdur, kdec, iolaps, \
    ifna, ifnb, itotdur [, iphs] [, ifmode] [, iskip]

ares fof2 xamp, xfund, xform, koct, kband, kris, kdur, kdec, iolaps, \
```

```

    ifna, ifnb, itotdur, kphs, kgliss [, iskip]

ares fog xamp, xdens, xtrans, aspd, koct, kband, kris, kdur, kdec, \
    iolaps, ifna, ifnb, itotdur [, iphs] [, itmode] [, iskip]

ares grain xamp, xpitch, xdens, kampoff, kpitchoff, kgdur, igfn, \
    iwfn, imgdur [, igrnd]

ares grain2 kcps, kfmd, kgdur, iovrlp, kfn, iwfn [, irpow] \
    [, iseed] [, imode]

ares grain3 kcps, kphs, kfmd, kpmf, kgdur, kdens, imaxovr, kfn, iwfn, \
    kfrpow, krpow [, iseed] [, imode]

ares granule xamp, ivoice, iratio, imode, ithd, ifn, ipshift, igskip, \
    igskip_os, ilength, kgap, igap_os, kgsz, igsz_os, iatt, idec \
    [, iseed] [, ipitch1] [, ipitch2] [, ipitch3] [, ipitch4] [, ifnenv]

a1 [, a2, a3, a4, a5, a6, a7, a8] partikkel agrainfreq, \
    kdistribution, idisttab, async, kenv2amt, ienv2tab, ienv_attack, \
    ienv_decay, ksustain_amount, ka_d_ratio, kduration, kamp, igainmasks, \
    kwavfreq, ksweepshape, iwavfreqstarttab, iwavfreqendtab, awavfm, \
    ifmampstab, kfmenv, icosine, ktraincps, knumpartials, kchroma, \
    ichannelmasks, krandmask, kwaveform1, kwaveform2, kwaveform3, \
    kwaveform4, iwaveamptab, asamplepos1, asamplepos2, asamplepos3, \
    asamplepos4, kwavekey1, kwavekey2, kwavekey3, kwavekey4, imax_grains \
    [, iopcode_id, ipanlaws]

kindex partikkelget kparameterindex, iopcode_id

partikkelset kparameterindex, kmaskindex, iopcode_id

asyncl [,aphase] partikkelsync iopcode_id

ares [, ac] sndwarp xamp, xtimewarp, xresample, ifn1, ibeg, iwsz, \
    irandw, ioverlap, ifn2, itimemode

arl, ar2 [,ac1] [, ac2] sndwarpst xamp, xtimewarp, xresample, ifn1, \
    ibeg, iwsz, irandw, ioverlap, ifn2, itimemode

asig syncgrain kamp, kfreq, kpitch, kgrsz, kprate, ifun1, \
    ifun2, iolaps

asig syncloop kamp, kfreq, kpitch, kgrsz, kprate, klstart, \
    klend, ifun1, ifun2, iolaps[,istart, iskip]

ar vosim kamp, kFund, kForm, kDecay, kPulseCount, kPulseFactor, ifn [, iskip]

```

### Signal Generators:Hyper Vectorial Synthesis.

```

hvs1 kx, inumParms, inumPointsX, iOutTab, iPositionsTab, iSnapTab [, iConfigTab]

hvs2 kx, ky, inumParms, inumPointsX, inumPointsY, iOutTab, iPositionsTab, iSnapTab [,
    iConfigTab]

hvs3 kx, ky, kz, inumParms, inumPointsX, inumPointsY, inumPointsZ, iOutTab, iPosition-
    sTab, iSnapTab [, iConfigTab]

```

### Signal Generators:Linear and Exponential Generators.

```

ky bpf kx, kx1, ky1, kx2, ..., kxn, kyn

ares cosseg ia, idur1, ib [, idur2] [, ic] [...]
kres cosseg ia, idur1, ib [, idur2] [, ic] [...]

ares cossegb ia, itim1, ib [, itim2] [, ic] [...]
kres cossegb ia, itim1, ib [, itim2] [, ic] [...]

ares cossegr ia, idur1, ib [, idur2] [, ic] [...], irel, iz
kres cossegr ia, idur1, ib [, idur2] [, ic] [...], irel, iz

kout expcurve kindex, ksteepness

ares expon ia, idur, ib
kres expon ia, idur, ib

ares expseg ia, idur1, ib [, idur2] [, ic] [...]
kres expseg ia, idur1, ib [, idur2] [, ic] [...]

ares expsega ia, idur1, ib [, idur2] [, ic] [...]

ares expsegb ia, itim1, ib [, itim2] [, ic] [...]
kres expsegb ia, itim1, ib [, itim2] [, ic] [...]

ares expsegba ia, itim1, ib [, itim2] [, ic] [...]

ares expsegr ia, idur1, ib [, idur2] [, ic] [...], irel, iz
kres expsegr ia, idur1, ib [, idur2] [, ic] [...], irel, iz

kout gainslider kindex

ares line ia, idur, ib
kres line ia, idur, ib

kout linlin kin, ksrclo, ksrchl, kdstlo, kdsthi

ares linseg ia, idur1, ib [, idur2] [, ic] [...]
kres linseg ia, idur1, ib [, idur2] [, ic] [...]

ares linsegb ia, itim1, ib [, itim2] [, ic] [...]
kres linsegb ia, itim1, ib [, itim2] [, ic] [...]

ares linsegr ia, idur1, ib [, idur2] [, ic] [...], irel, iz
kres linsegr ia, idur1, ib [, idur2] [, ic] [...], irel, iz

kout logcurve kindex, ksteepness

ksig loopseg kfreq, ktrig, iphase, kvalue0, ktime0 [, kvalue1] [, ktime1] \
[, kvalue2] [, ktime2][...]

ksig loopsegp kphase, kvalue0, kdur0, kvalue1 \
[, kdur1, ... , kdurN-1, kvalueN]

ksig looptseg kfreq, ktrig, iphase, kvalue0, ktype0, ktime0, [, kvalue1] [,ktype1] [,
ktime1] \
[, kvalue2] [,ktype2] [, ktime2] [...] [, kvalueN] [,ktypeN] [, ktimeN]

ksig loopxseg kfreq, ktrig, iphase, ktime0, kvalue0 [, ktime1] [, kvalue1] \
[, ktime2] [, kvalue2] [...]

ksig lpshold kfreq, ktrig, iphase, ktime0, kvalue0 [, kvalue1] [, ktime1] [, kvalue2]
[, ktime2] [...]

ksig lpsholdp kphase, kvalue0, ktime0 [, kvalue1] [, ktime1] \
[, kvalue2] [, ktime2] [...]

```

```
kscl scale kinput, kmax, kmin

ares transeg ia, idur, itype, ib [, idur2] [, itype] [, ic] ...
kres transeg ia, idur, itype, ib [, idur2] [, itype] [, ic] ...

ares transegb ia, itim, itype, ib [, itim2] [, itype] [, ic] ...
kres transegb ia, itim, itype, ib [, itim2] [, itype] [, ic] ...

ares transegr ia, idur, itype, ib [, idur2] [, itype] [, ic] ...
kres transegr ia, idur, itype, ib [, idur2] [, itype] [, ic] ...

kout xyscale kx, ky, k00, k10, k01, k11
```

### Signal Generators:Envelope Generators.

```
ares adsr iatt, idec, islev, irel [, idel]
kres adsr iatt, idec, islev, irel [, idel]

ares envlpx xamp, irise, idur, idec, ifn, iatss, iatdec [, ixmod]
kres envlpx kamp, irise, idur, idec, ifn, iatss, iatdec [, ixmod]

ares envlpxr xamp, irise, idec, ifn, iatss, iatdec [, ixmod] [,irind]
kres envlpxr kamp, irise, idec, ifn, iatss, iatdec [, ixmod] [,irind]

ares linen xamp, irise, idur, idec
kres linen kamp, irise, idur, idec

ares linenr xamp, irise, idec, iatdec
kres linenr kamp, irise, idec, iatdec

ares madsr iatt, idec, islev, irel [, idel] [, ireltim]
kres madsr iatt, idec, islev, irel [, idel] [, ireltim]

ares mxadsr iatt, idec, islev, irel [, idel] [, ireltim]
kres mxadsr iatt, idec, islev, irel [, idel] [, ireltim]

ares xadsr iatt, idec, islev, irel [, idel]
kres xadsr iatt, idec, islev, irel [, idel]
```

### Signal Generators:Models and Emulations.

```
ares bamboo kamp, idettack [, inum] [, idamp] [, imaxshake] [, ifreq] \
    [, ifreq1] [, ifreq2]

ares barmodel kbcL, kbcR, iK, ib, kscan, iT30, ipos, ivel, iwid

ares cabasa iamp, idettack [, inum] [, idamp] [, imaxshake]

aI3, aV2, aV1 chuap kL, kR0, kC1, kG, kGa, kGb, kE, kC2, iI3, iV2, iV1, ktime_step

ares crunch iamp, idettack [, inum] [, idamp] [, imaxshake]

ares dripwater kamp, idettack [, inum] [, idamp] [, imaxshake] [, ifreq] \
    [, ifreq1] [, ifreq2]

ares gandy kamp, kampdist, kdurdist, kadpar, kddpar, kminfreq, kmaxfreq, \
    kampscl, kdurscl [, initcps] [, knum]
kres gandy kamp, kampdist, kdurdist, kadpar, kddpar, kminfreq, kmaxfreq, \
    kampscl, kdurscl [, initcps] [, knum]
```

```

ares gendyc kamp, kampdist, kdurdist, kadpar, kddpar, kminfreq, kmaxfreq, \
    kampscl, kdurscl [, initcps] [, knum]
kres gendyc kamp, kampdist, kdurdist, kadpar, kddpar, kminfreq, kmaxfreq, \
    kampscl, kdurscl [, initcps] [, knum]

ares gendyx kamp, kampdist, kdurdist, kadpar, kddpar, kminfreq, kmaxfreq, \
    kampscl, kdurscl, kcurveup, kcurvedown [, initcps] [, knum]
kres gendyx kamp, kampdist, kdurdist, kadpar, kddpar, kminfreq, kmaxfreq, \
    kampscl, kdurscl, kcurveup, kcurvedown [, initcps] [, knum]

ares gogobel kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivfn

ares guiro kamp, idettack [, inum] [, idamp] [, imaxshake] [, ifreq] [, ifreq1]

ax, ay, az lorenz ksv, krv, kbv, kh, ix, iy, iz, iskip [, iskipinit]

kiter, koutrig mandel ktrig, kx, ky, kmaxIter

ares mandol kamp, kfreq, kpluck, kdetune, kgain, ksize \
    [, ifn] [, iminfreq]

ares marimba kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivibfn, idec \
    [, idoubles] [, itriples]

ares moog kamp, kfreq, kfiltq, kfiltrate, kvibf, kvamp, iafn, iwfn, ivfn

ax, ay, az planet kmass1, kmass2, ksep, ix, iy, iz, ivx, ivy, ivz, idelta \
    [, ifriction] [, iskip]

ares prepiano ifreq, iNS, iD, iK, \
    iT30,iB, kbcl, kbcr, imass, ihvfreq, iinit, ipos, ivel, isfreq, \
    isspread[, irattles, irubbers]
al,ar prepiano ifreq, iNS, iD, iK, \
    iT30,iB, kbcl, kbcr, imass, ihvfreq, iinit, ipos, ivel, isfreq, \
    isspread[, irattles, irubbers]

ares sandpaper iamp, idettack [, inum] [, idamp] [, imaxshake]

ares sekere iamp, idettack [, inum] [, idamp] [, imaxshake]

ares shaker kamp, kfreq, kbeans, kdamp, ktimes [, idecay]

ares sleighbells kamp, idettack [, inum] [, idamp] [, imaxshake] [, ifreq] \
    [, ifreq1] [, ifreq2]

ares stix iamp, idettack [, inum] [, idamp] [, imaxshake]

ares tambourine kamp, idettack [, inum] [, idamp] [, imaxshake] [, ifreq] \
    [, ifreq1] [, ifreq2]

ares vibes kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivibfn, idec

ares voice kamp, kfreq, kphoneme, kform, kvibf, kvamp, ifn, ivfn

```

### Signal Generators:Phasors.

```

ares phasor xcps [, iphs]
kres phasor kcps [, iphs]

ares phasorbnk xcps, kndx, icnt [, iphs]
kres phasorbnk kcps, kndx, icnt [, iphs]

aindex sc_phasor xtrig, xrate, kstart, kend [, kresetPos]

```

kindex **sc\_phasor** xtrig, xrate, kstart, kend [, kresetPos]  
 (end - start) \* freq / sr

aphase, asyncout **syncphasor** xcps, asyncin, [, iphs]

### Signal Generators:Random (Noise) Generators.

ares **betarand** krange, kalpha, kbeta  
 ires **betarand** krange, kalpha, kbeta  
 kres **betarand** krange, kalpha, kbeta

ares **bexprnd** krange  
 ires **bexprnd** krange  
 kres **bexprnd** krange

ares **cauchy** kalpha  
 ires **cauchy** kalpha  
 kres **cauchy** kalpha

ares **cauchy** klambda, xamp, xcps  
 ires **cauchy** klambda, xamp, xcps  
 kres **cauchy** klambda, xamp, xcps

aout **cuserrnd** kmin, kmax, ktableNum  
 iout **cuserrnd** imin, imax, itableNum  
 kout **cuserrnd** kmin, kmax, ktableNum

aout **duserrnd** ktableNum  
 iout **duserrnd** itableNum  
 kout **duserrnd** ktableNum

ares **dust** kamp, kdensity  
 kres **dust** kamp, kdensity

ares **dust2** kamp, kdensity  
 kres **dust2** kamp, kdensity

ares **exprand** klambda  
 ires **exprand** klambda  
 kres **exprand** klambda

ares **exprandi** klambda, xamp, xcps  
 ires **exprandi** klambda, xamp, xcps  
 kres **exprandi** klambda, xamp, xcps

ares **fractalnoise** kamp, kbeta

ares **gauss** krange  
 ires **gauss** krange  
 kres **gauss** krange

ares **gaussi** krange, xamp, xcps  
 ires **gaussi** krange, xamp, xcps  
 kres **gaussi** krange, xamp, xcps

ares **gausstrig** kamp, kcps, kdev [, imode] [, ifrst1]  
 kres **gausstrig** kamp, kcps, kdev [, imode] [, ifrst1]

ians **getseed**  
 kans **getseed**

kout **jitter** kamp, kcpsMin, kcpsMax

```

kout jitter2 ktotamp, kamp1, kcps1,
    kamp2, kcps2, kamp3, kcps3[ , iopt]

ares jspline xamp, kcpsMin, kcpsMax
kres jspline kamp, kcpsMin, kcpsMax

ares linrand krange
ires linrand krange
kres linrand krange

ares noise xamp, kbeta

ares pcauchy kalpha
ires pcauchy kalpha
kres pcauchy kalpha

ares pinker

ares pinkish xin [, imethod] [, inumbands] [, iseed] [, iskip]

ares poisson klambda
ires poisson klambda
kres poisson klambda

ares rand xamp [, iseed] [, isel] [, ioffset]
kres rand xamp [, iseed] [, isel] [, ioffset]

ares randh xamp, xcps [, iseed] [, isize] [, ioffset]
kres randh kamp, kcps [, iseed] [, isize] [, ioffset]

ares randi xamp, xcps [, iseed] [, isize] [, ioffset]
kres randi kamp, kcps [, iseed] [, isize] [, ioffset]

ares random kmin, kmax
ires random imin, imax
kres random kmin, kmax

ares randomh kmin, kmax, xcps [,imode] [,ifirstval]
kres randomh kmin, kmax, kcps [,imode] [,ifirstval]

ares randomi kmin, kmax, xcps [,imode] [,ifirstval]
kres randomi kmin, kmax, kcps [,imode] [,ifirstval]

ax rnd31 kscl, krpow [, iseed]
ix rnd31 iscl, irpow [, iseed]
kx rnd31 kscl, krpow [, iseed]

ares rspline xrangeMin, xrangeMax, kcpsMin, kcpsMax
kres rspline krangeMin, krangeMax, kcpsMin, kcpsMax

seed ival

kout trandom ktrig, kmin, kmax

ares trirand krange
ires trirand krange
kres trirand krange

ares unirand krange
ires unirand krange
kres unirand krange

ax urandom [imin, imax]
ix urandom [imin, imax]
kx urandom [imin, imax]

aout = urd(ktableNum)

```

```
iout = urd(itableNum)
kout = urd(ktableNum)

ares weibull ksigma, ktau
ires weibull ksigma, ktau
kres weibull ksigma, ktau
```

### Signal Generators:Sample Playback.

```
a1 bbcutm asource, ibps, isubdiv, ibarlength, iphrasebars, inumrepeats \
    [, istutterspeed] [, istutterchance] [, ienvchoice ]

a1,a2 bbcuts asource1, asource2, ibps, isubdiv, ibarlength, iphrasebars, \
    inumrepeats [, istutterspeed] [, istutterchance] [, ienvchoice]

asig1[, asig2] flooper kamp, kpitch, istart, idur, ifad, ifn

asig1[,asig2] flooper2 kamp, kpitch, kloopstart, kloopend, kcrossfade, ifn \
    [, istart, imode, ifenv, iskip]

aleft, aright fluidAllOut

fluidCCi iEngineNumber, iChannelNumber, iControllerNumber, iValue

fluidCCK iEngineNumber, iChannelNumber, iControllerNumber, kValue

fluidControl ienginenum, kstatus, kchannel, kdata1, kdata2

ienginenum fluidEngine [iReverbEnabled] [, iChorusEnabled] [,iNumChannels] [, iPolypho-
ny]

isfnum fluidLoad soundfont, ienginenum[, ilistpresets]

fluidNote ienginenum, ichannelnum, imidikey, imidivel

aleft, aright fluidOut ienginenum

fluidProgramSelect ienginenum, ichannelnum, isfnum, ibanknum, ipresetnum

fluidSetInterpMethod ienginenum, ichannelnum, iInterpMethod

ar1 [,ar2] loscil xamp, kcps, ifn [, ibas] [, imod1] [, ibeg1] [, iend1] \
    [, imod2] [, ibeg2] [, iend2]

ar1 [,ar2] loscil3 xamp, kcps, ifn [, ibas] [, imod1] [, ibeg1] [, iend1] \
    [, imod2] [, ibeg2] [, iend2]

ar1 [, ar2, ar3, ar4, ar5, ar6, ar7, ar8, ar9, ar10, ar11, ar12, ar13, ar14, \
    ar15, ar16] loscilx xamp, kcps, ifn \
    [, iwsiz, ibas, istr, imod, ibeg, iend]

ares lphaser xtrns [, ilps] [, ilpe] [, imode] [, istr] [, istor]

ares lposcil kamp, kfregratio, kloop, kend, ifn [, iphs]

ares lposcil3 kamp, kfregratio, kloop, kend, ifn [, iphs]

ar lposcila aamp, kfregratio, kloop, kend, ift [,iphs]

ar1, ar2 lposcilsa aamp, kfregratio, kloop, kend, ift [,iphs]

ar1, ar2 lposcilsa2 aamp, kfregratio, kloop, kend, ift [,iphs]
```

```

sfilist ifilhandle

ar1, ar2 sfinstr ivel, inotenum, xamp, xfreq, instrnum, ifilhandle \
    [, iflag] [, ioffset]

ar1, ar2 sfinstr3 ivel, inotenum, xamp, xfreq, instrnum, ifilhandle \
    [, iflag] [, ioffset]

ares sfinstr3m ivel, inotenum, xamp, xfreq, instrnum, ifilhandle \
    [, iflag] [, ioffset]

ares sfinstrm ivel, inotenum, xamp, xfreq, instrnum, ifilhandle \
    [, iflag] [, ioffset]

ir sfload "filename"

ar1, ar2 sflooper ivel, inotenum, kamp, kpitch, ipreindex, kloopstart, kloopend,
    kcrossfade \
    [, istart, imode, ifenv, iskip]

sfpassign istartindex, ifilhandle[, imsgs]

ar1, ar2 sfplay ivel, inotenum, xamp, xfreq, ipreindex [, iflag] [, ioffset] [, ienv]

ar1, ar2 sfplay3 ivel, inotenum, xamp, xfreq, ipreindex [, iflag] [, ioffset] [, ienv]

ares sfplay3m ivel, inotenum, xamp, xfreq, ipreindex [, iflag] [, ioffset] [, ienv]

ares sfplaym ivel, inotenum, xamp, xfreq, ipreindex [, iflag] [, ioffset] [, ienv]

sfplist ifilhandle

ir sfpreset iprog, ibank, ifilhandle, ipreindex

asig, krec sndloop ain, kpitch, ktrig, idur, ifad

ares waveset ain, krep [, ilen]

```

### Signal Generators:Scanned Synthesis.

```

scanhammer isrc, idst, ipos, imult

ares scans kamp, kfreq, ifn, id [, iorder]

aout scantable kamp, kpch, ipos, imass, istiff, idamp, ivel

scanu init, irate, ifnvel, ifnmass, ifnstif, ifncentr, ifndamp, kmass, \
    kstif, kcentr, kdamp, ileft,  iright, kpos, kstrngth, ain, idisp, id

kpos, kvel xscanmap iscan, kamp, kvamp [, iwhich]

ares xscans kamp, kfreq, ifntraj, id [, iorder]

xscansmap kpos, kvel, iscan, kamp, kvamp [, iwhich]

xscanu init, irate, ifnvel, ifnmass, ifnstif, ifncentr, ifndamp, kmass, \
    kstif, kcentr, kdamp, ileft,  iright, kpos, kstrngth, ain, idisp, id

```

### Signal Generators:STK Opcodes.

asignal **STKBandedWG** ifrequency, iamplitude, [kpress, kv1[, kmot, kv2[, klfo, kv3[, kl-  
fodepth, kv4[, kvel, kv5[, kstrk, kv6[, kinstr, kv7]]]]]]]]]

asignal **STKBeeThree** ifrequency, iamplitude, [kop4, kv1[, kop3, kv2[, klfo, kv3[, kl-  
fodepth, kv4[, kadsr, kv5]]]]]]]

asignal **STKBlowBot1** ifrequency, iamplitude, [knoise, kv1[, klfo, kv2[, klfodepth, kv3[,  
kvol, kv4]]]]]

asignal **STKBlowHole** ifrequency, iamplitude, [kreed, kv1[, knoise, kv2[, khole, kv3[,  
kreg, kv4[, kbreath, kv5]]]]]]]

asignal **STKBowed** ifrequency, iamplitude, [kpress, kv1[, kpos, kv2[, klfo, kv3[, kl-  
fodepth, kv4[, kvol, kv5]]]]]]]

asignal **STKBrass** ifrequency, iamplitude, [klip, kv1[, kslide, kv2[, klfo, kv3[, kl-  
fodepth, kv4[, kvol, kv5]]]]]]]

asignal **STKClarinet** ifrequency, iamplitude, [kstiff, kv1[, knoise, kv2[, klfo, kv3[,  
klfodepth, kv4[, kbreath, kv5]]]]]]]

asignal **STKDrummer** ifrequency, iamplitude

asignal **STKFMVoices** ifrequency, iamplitude, [kvowel, kv1[, kspec, kv2[, klfo, kv3[, kl-  
fodepth, kv4[, kadsr, kv5]]]]]]]

asignal **STKFlute** ifrequency, iamplitude, [kjet, kv1[, knoise, kv2[, klfo, kv3[, kl-  
fodepth, kv4[, kbreath, kv5]]]]]]]

asignal **STKHevyMet1** ifrequency, iamplitude, [kmod, kv1[, kcross, kv2[, klfo, kv3[, kl-  
fodepth, kv4[, kadsr, kv5]]]]]]]

asignal **STKMandolin** ifrequency, iamplitude, [kbody, kv1[, kpos, kv2[, ksus, kv3[, kde-  
tune, kv4[, kmic, kv5]]]]]]]

asignal **STKModalBar** ifrequency, iamplitude, [khard, kv1[, kpos, kv2[, klfo, kv3[, kl-  
fodepth, kv4[, kmix, kv5[, kvol, kv6[, kinstr, kv7]]]]]]]]]

asignal **STKMoog** ifrequency, iamplitude, [kq, kv1[, krate, kv2[, klfo, kv3[, klfodepth,  
kv4[, kvol, kv5]]]]]]]

asignal **STKPercFlut** ifrequency, iamplitude, [kmod, kv1[, kcross, kv2[, klfo, kv3[, kl-  
fodepth, kv4[, kadsr, kv5]]]]]]]

asignal **STKPlucked** ifrequency, iamplitude

asignal **STKResonate** ifrequency, iamplitude, [kfreq, kv1[, kpole, kv2[, notch, kv3[,  
kzero, kv4[, kenv, kv5]]]]]]]

asignal **STKRhody** ifrequency, iamplitude, [kmod, kv1[, kcross, kv2[, klfo, kv3[, kl-  
fodepth, kv4[, kadsr, kv5]]]]]]]

asignal **STKSaxofony** ifrequency, iamplitude, [kstiff, kv1[, kapert, kv2[, kblow, kv3[,  
knoise, kv4[, klfo, kv5[, klfodepth, kv6[, kbreath, kv7]]]]]]]]]]]

asignal **STKShakers** ifrequency, iamplitude, [kenerg, kv1[, kdecay, kv2[, kshake, kv3[,  
knum, kv4[, kres, kv5[, kinstr, kv6]]]]]]]]]

asignal **STKSimple** ifrequency, iamplitude, [kpos, kv1[, kcross, kv2[, kenv, kv3[, kgain,  
kv4]]]]]

asignal **STKSitar** ifrequency, iamplitude

asignal **STKStifKarp** ifrequency, iamplitude, [kpos, kv1[, ksus, kv2[, kstretch, kv3]]]

asignal **STKTubeBell** ifrequency, iamplitude, [kmod, kv1[, kcross, kv2[, klfo, kv3[, kl-  
fodepth, kv4[, kadsr, kv5]]]]]]]

asignal **STKVoicForm** ifrequency, iamplitude, [kmix, kv1[, ksel, kv2[, klfo, kv3[, kl-  
fodepth, kv4[, kcloud, kv5]]]]]

asignal **STKWhistle** ifrequency, iamplitude, [kmod, kv1[, knoise, kv2[, kfipfreq, kv3[,  
kfipgain, kv4[, kvol, kv5]]]]]

asignal **STKWurley** ifrequency, iamplitude, [kmod, kv1[, kcross, kv2[, klfo, kv3[, kl-  
fodepth, kv4[, kadsr, kv5]]]]]

### Signal Generators:Table Access.

kres **oscill** idel, kamp, idur [, ifn]

kres **oscilli** idel, kamp, idur [, ifn]

ares **ptable** andx, ifn [, ixmode] [, ixoff] [, iwrap]  
ires **ptable** indx, ifn [, ixmode] [, ixoff] [, iwrap]  
kres **ptable** kndx, ifn [, ixmode] [, ixoff] [, iwrap]

ares **ptable3** andx, ifn [, ixmode] [, ixoff] [, iwrap]  
ires **ptable3** indx, ifn [, ixmode] [, ixoff] [, iwrap]  
kres **ptable3** kndx, ifn [, ixmode] [, ixoff] [, iwrap]

ares **ptablei** andx, ifn [, ixmode] [, ixoff] [, iwrap]  
ires **ptablei** indx, ifn [, ixmode] [, ixoff] [, iwrap]  
kres **ptablei** kndx, ifn [, ixmode] [, ixoff] [, iwrap]

ir **tab\_i** indx, ifn[, ixmode]  
kr **tab** kndx, ifn[, ixmode]  
ar **tab** xndx, ifn[, ixmode]  
**tabw\_i** isig, indx, ifn [,ixmode]  
**tabw** ksig, kndx, ifn [,ixmode]  
**tabw** asig, andx, ifn [,ixmode]

ares **table** andx, ifn [, ixmode] [, ixoff] [, iwrap]  
ires **table** indx, ifn [, ixmode] [, ixoff] [, iwrap]  
kres **table** kndx, ifn [, ixmode] [, ixoff] [, iwrap]

ares **table3** andx, ifn [, ixmode] [, ixoff] [, iwrap]  
ires **table3** indx, ifn [, ixmode] [, ixoff] [, iwrap]  
kres **table3** kndx, ifn [, ixmode] [, ixoff] [, iwrap]

ares **tablei** andx, ifn [, ixmode] [, ixoff] [, iwrap]  
ires **tablei** indx, ifn [, ixmode] [, ixoff] [, iwrap]  
kres **tablei** kndx, ifn [, ixmode] [, ixoff] [, iwrap]

### Signal Generators:Wave Terrain Synthesis.

aout **wterrain** kamp, kpch, k\_xcenter, k\_ycenter, k\_xradius, k\_yradius, \  
itabx, itaby

### Signal Generators:Waveguide Physical Modeling.

ares **pluck** kamp, kcps, icps, ifn, imeth [, iparm1] [, iparm2]

ares **repluck** iplk, kamp, icps, kpick, krefl, excite

```
ares streson asig, kfr, kfdbgain

ares wgbow kamp, kfreq, kpres, krat, kvibf, kvamp \
    [, ifn] [, iminfreq]

ares wgbowedbar kamp, kfreq, kpos, kbowpres, kgain [, iconst] [, itvel] \
    [, ibowpos] [, ilow]

ares wgbrass kamp, kfreq, ktens, iatt, kvibf, kvamp \
    [, ifn] [, iminfreq]

ares wgclar kamp, kfreq, kstiff, \
    iatt, idetk, kngain, kvibf, kvamp [, ifn] [, iminfreq]

ares wgflute kamp, kfreq, kjet, iatt,
    idetk, kngain, kvibf, kvamp [, ifn] [, iminfreq] [, ijetrfl] [, iendrf]

ares wgpluck icps, iamp, kpick, iplk, idamp, ifilt, axcite

ares wgpluck2 iplk, kamp, icps, kpick, krefl
```

### Signal I/O:File I/O.

```
dumpk ksig, ifilename, iformat, iprd

dumpk2 ksig1, ksig2, ifilename, iformat, iprd

dumpk3 ksig1, ksig2, ksig3, ifilename, iformat, iprd

dumpk4 ksig1, ksig2, ksig3, ksig4, ifilename, iformat, iprd

ficlose ihandle
ficlose Sfilename

fin ifilename, iskipframes, iformat, ain1 [, ain2] [, ain3] [,...]
fin ifilename, iskipframes, iformat, arr[]

fini ifilename, iskipframes, iformat, in1 [, in2] [, in3] [, ...]

fink ifilename, iskipframes, iformat, kin1 [, kin2] [, kin3] [,...]

ihandle fiopen ifilename, imode

fout ifilename, iformat, aout1 [, aout2, aout3,...,aoutN]
fout ifilename, iformat, array[]

fouti ihandle, iformat, iflag, iout1 [, iout2, iout3,...,ioutN]

foutir ihandle, iformat, iflag, iout1 [, iout2, iout3,...,ioutN]

foutk ifilename, iformat, kout1 [, kout2, kout3,...,koutN]

fprintks "filename", "string", [, kval1] [, kval2] [...]

fprints "filename", "string" [, ival1] [, ival2] [...]

xout1[, xout2, xout3, ..., xoutN] hdf5read ifilename, ivariablename1[, ivariablename2,
    ivariablename3, ..., ivariablenameN]

hdf5write ifilename, xout1[, xout2, xout3, ..., xoutN]

Sres, kline readf ifilename
```

```
Sres, iline readfi ifilename
kres readk ifilename, iformat, iprd
kr1, kr2 readk2 ifilename, iformat, iprd
kr1, kr2, kr3 readk3 ifilename, iformat, iprd
kr1, kr2, kr3, kr4 readk4 ifilename, iformat, iprd
```

### Signal I/O:Signal Input.

```
ar1 [, ar2 [, ar3 [, ... arN]]] diskin ifilcod[, kpitch[, iskiptim \
    [, iwraparound[, iformat[, iskipinit]]]]]
ar1[] diskin ifilcod[, kpitch[, iskiptim \
    [, iwraparound[, iformat[, iskipinit]]]]]

a1[, a2[, ... aN]] diskin2 ifilcod[, kpitch[, iskiptim \
    [, iwrap[, iformat[, iwsizel[, ibufsize[, iskipinit]]]]]]]
ar1[] diskin2 ifilcod[, kpitch[, iskiptim \
    [, iwrap[, iformat[, iwsizel[, ibufsize[, iskipinit]]]]]]]

ar1 in
aarray in

ar1, ar2, ar3, ar4, ar5, ar6, ar7, ar8, ar9, ar10, ar11, ar12, ar13, ar14, \
    ar15, ar16, ar17, ar18, ar19, ar20, ar21, ar22, ar23, ar24, ar25, ar26, \
    ar27, ar28, ar29, ar30, ar31, ar32 in32

ain1[, ...] inch kchan1[,...]

ar1, ar2, ar3, ar4, ar5, ar6 inh

ar1, ar2, ar3, ar4, ar5, ar6, ar7, ar8 ino

ar1, ar2, ar3, a4 inq

inrg kstart, ain1 [,ain2, ain3, ..., ainN]

ar1, ar2 ins

ivaluel invalue "channel name"
kvaluel invalue "channel name"
Sname invalue "channel name"

ar1, ar2, ar3, ar4, ar5, ar6, ar7, ar8, ar9, ar10, ar11, ar12, \
    ar13, ar14, ar15, ar16 inx

inz ksig1

ar1, ar2 mp3in ifilcod[, iskptim, iformat, iskipinit, ibufsize]
ar1 mp3in ifilcod[, iskptim, iformat, iskipinit, ibufsize]

ar1[, ar2[, ar3[, ... a24]]] soundin ifilcod [, iskptim] [, iformat] \
    [, iskipinit] [, ibufsize]
```

### Signal I/O:Signal Output.

```
mdelay kstatus, kchan, kd1, kd2, kdelay
```

```

aout1 [,aout2 ... aoutX] monitor
aarra monitor

out asig1[, asig2,...]
out aarray

out32 asig1, asig2, asig3, asig4, asig5, asig6, asig7, asig8, asig10, \
      asig11, asig12, asig13, asig14, asig15, asig16, asig17, asig18, \
      asig19, asig20, asig21, asig22, asig23, asig24, asig25, asig26, \
      asig27, asig28, asig29, asig30, asig31, asig32

outc asig1 [, asig2] [...]

outch kchan1, asig1 [, kchan2] [, asig2] [...]

outh asig1, asig2, asig3, asig4, asig5, asig6

outo asig1, asig2, asig3, asig4, asig5, asig6, asig7, asig8

outq asig1, asig2, asig3, asig4

outq1 asig
outq2 asig
outq3 asig
outq4 asig

outrg kstart, aout1 [,aout2, aout3, ..., aoutN]

outs asig1, asig2

outs1 asig
outs2 asig

outvalue "channel name", ivalue
outvalue "channel name", kvalue
outvalue "channel name", "string"

outx asig1, asig2, asig3, asig4, asig5, asig6, asig7, asig8, \
      asig9, asig10, asig11, asig12, asig13, asig14, asig15, asig16

outz ksig1

soundout asig1, ifilcod [, iformat]

soundouts asig1, asigr, ifilcod [, iformat]

```

### Signal I/O:Software Bus.

```

kval chani kchan
aval chani kchan

chano kval, kchan
chano aval, kchan

chn_k Sname, imode[, itype, idflt, imin, ima, ix, iy, iwidth, iheight, Sattributes]
chn_a Sname, imode
chn_s Sname, imode

chnclear Sname

```

```

gival chnexport Sname, imode[, itype, idflt, imin, imax]
gkval chnexport Sname, imode[, itype, idflt, imin, imax]
gaval chnexport Sname, imode
gSval chnexport Sname, imode

ival chnget Sname
kval chnget Sname
aval chnget Sname
Sval chnget Sname

chnmix aval, Sname

itype, imode, ictltype, idflt, imin, imax chnparams Sname

chnset ival, Sname
chnset kval, Sname
chnset aval, Sname
chnset Sval, Sname

setksmps ikmps

xinarg1 [, xinarg2] ... [xinargN] xin

xout xoutarg1 [, xoutarg2] ... [, xoutargN]

```

### Signal I/O:Printing and Display.

```

dispffft xsig, iprd, iwsiz [, iwtyp] [, idbout] [, iwtflg] [,imin] [,imax]

display xsig, iprd [, inprds] [, iwtflg]

flashtxt iwhich, String

print iarg [, iarg1] [, iarg2] [...]

printf_i Sfmt, itrig, [iarg1[, iarg2[, ... ]]]
printf Sfmt, ktrig, [xarg1[, xarg2[, ... ]]]

printk itime, kval [, ispace]

printk2 kvar [, inumspaces]

printks "string", itime [, kval1] [, kval2] [...]

printks2 "string", kval

prints "string" [, kval1] [, kval2] [...]

```

### Signal I/O:Soundfile Queries.

```

ir filebit ifilcod [, iallowraw]

ir filelen ifilcod, [iallowraw]

ir filenchnls ifilcod [, iallowraw]

ir filepeak ifilcod [, ichnl]

ir filesr ifilcod [, iallowraw]

```

ir **filevalid** ifilcod

ir **mp3len** ifilcod

### Signal Modifiers:Amplitude Modifiers.

ares **balance** asig, acomp [, ihp] [, iskip]

ares **clip** asig, imeth, ilimit [, iarg]

ar **compress** aasig, acsig, kthresh, kloknee, khiknee, kratio, katt, krel, ilook

ar **compress2** aasig, acsig, kthresh, kloknee, khiknee, kratio, katt, krel, ilook

ares **dam** asig, kthreshold, icomp1, icomp2, irtime, iftime

ares **gain** asig, krms [, ihp] [, iskip]

### Signal Modifiers:Convolution and Morphing.

ar1 [, ar2] [, ar3] [, ar4] **convolve** ain, ifilcod [, ichannel]

ares **cross2** ain1, ain2, isize, ioverlap, iwin, kbias

ares **dconv** asig, isize, ifn

a1[, a2[, a3[, ... a8]]] **ftconv** ain, ift, iplen[, iskipsamples \  
 [, iirlen[, iskipinit]]]

**ftmorf** kftndx, iftn, iresfn

ares **liveconv** ain, ift, iplen, kupdate, kclear

ar1 [, ar2] [, ar3] [, ar4] **pconvolve** ain, ifilcod [, ipartitionsizes, ichannel]

ares **tvconv** asig1, asig2, xfreez1,  
 xfreez2, iparts, ifils

### Signal Modifiers:Delay.

ares **delay** asig, idlt [, iskip]

ares **delay1** asig [, iskip]

kr **delayk** ksig, idel[, imode]

kr **vdel\_k** ksig, kdel, imdel[, imode]

ares **delayr** idlt [, iskip]

**delayw** asig

ares **deltap** kdlt

ares **deltap3** xdlt

ares **deltapi** xdlt

```

ares deltapn xnumsamps

aout deltapx adel, iwsizex

deltapxw ain, adel, iwsizex

ares multitap asig [, itime1, igain1] [, itime2, igain2] [...]

ares vdelay asig, adel, imaxdel [, iskip]

ares vdelay3 asig, adel, imaxdel [, iskip]

aout vdelayx ain, adl, imd, iws [, ist]

aout1, aout2, aout3, aout4 vdelayxq ain1, ain2, ain3, ain4, adl, imd, iws [, ist]

aout1, aout2 vdelayxs ain1, ain2, adl, imd, iws [, ist]

aout vdelayxw ain, adl, imd, iws [, ist]

aout1, aout2, aout3, aout4 vdelayxwq ain1, ain2, ain3, ain4, adl, \
    imd, iws [, ist]

aout1, aout2 vdelayxws ain1, ain2, adl, imd, iws [, ist]

```

### Signal Modifiers:Panning and Spatialization.

```

aol, ao2 bformdec isetup, aw, ax, ay, az [, ar, as, at, au, av \
    [, abk, al, am, an, ao, ap, aq]]
aol, ao2, ao3, ao4 bformdec isetup, aw, ax, ay, az [, ar, as, at, \
    au, av [, abk, al, am, an, ao, ap, aq]]
aol, ao2, ao3, ao4, ao5 bformdec isetup, aw, ax, ay, az [, ar, as, \
    at, au, av [, abk, al, am, an, ao, ap, aq]]
aol, ao2, ao3, ao4, ao5, ao6, ao7, ao8 bformdec isetup, aw, ax, ay, az \
    [, ar, as, at, au, av [, abk, al, am, an, ao, ap, aq]]]

aol, ao2 bformdec1 isetup, aw, ax, ay, az [, ar, as, at, au, av \
    [, abk, al, am, an, ao, ap, aq]]
aol, ao2, ao3, ao4 bformdec1 isetup, aw, ax, ay, az [, ar, as, at, \
    au, av [, abk, al, am, an, ao, ap, aq]]
aol, ao2, ao3, ao4, ao5 bformdec1 isetup, aw, ax, ay, az [, ar, as, \
    at, au, av [, abk, al, am, an, ao, ap, aq]]
aol, ao2, ao3, ao4, ao5, ao6, ao7, ao8 bformdec1 isetup, aw, ax, ay, az \
    [, ar, as, at, au, av [, abk, al, am, an, ao, ap, aq]]]
aout[] bformdec1 isetup, abform[]

aw, ax, ay, az bformenc asig, kalpha, kbeta, kord0, kord1
aw, ax, ay, az, ar, as, at, au, av bformenc asig, kalpha, kbeta, \
    kord0, kord1, kord2
aw, ax, ay, az, ar, as, at, au, av, ak, al, am, an, ao, ap, aq bformenc \
    asig, kalpha, kbeta, kord0, kord1, kord2, kord3

aw, ax, ay, az bformenc1 asig, kalpha, kbeta
aw, ax, ay, az, ar, as, at, au, av bformenc1 asig, kalpha, kbeta
aw, ax, ay, az, ar, as, at, au, av, ak, al, am, an, ao, ap, aq bformenc1 \
    asig, kalpha, kbeta
aarray[] bformenc1 asig, kalpha, kbeta

aleft, aright, irt60low, irt60high, imfp hrtfearly asrc, ksrx, ksry, ksrcz, klstnrx,
    klstnry, klstnrz, \
    ifile1, ifiler, ideoform [, ifade, isr, iorder, ithreed, kheadrot, iroomx, iroomy,
    iroomz, iwallhigh, \

```

```

        iwalllow, iwallgain1, iwallgain2, iwallgain3, ifloorhigh, ifloorlow, ifloorgain1,
        ifloorgain2, \
        ifloorgain3, iceilinghigh, iceilinglow, iceilinggain1, iceilinggain2, iceiling-
        gain3]

aleft, aright hrtfer asig, kaz, kelev, HRTFcompact

aleft, aright hrtfmove asrc, kAz, kElev, ifile1, ifiler [, imode, ifade, isr]

aleft, aright hrtfmove2 asrc, kAz, kElev, ifile1, ifiler [,ioverlap, iradius, isr]

aleft, aright, idel hrtfverb asrc, ilowrt60, ihighrt60, ifile1, ifiler [,isr, imfp,
        iorder]

        aleft, aright hrtfstat asrc, iAz, iElev, ifile1, ifiler [,iradius, isr]

a1, a2 locsend
a1, a2, a3, a4 locsend

a1, a2 locsig asig, kdegree, kdistance, kreverbsend
a1, a2, a3, a4 locsig asig, kdegree, kdistance, kreverbsend

a1, a2, a3, a4 pan asig, kx, ky, ifn [, imode] [, ioffset]

a1, a2 pan2 asig, xp [, imode]

a1, a2, a3, a4 space asig, ifn, ktime, kreverbsend, kx, ky

aW, aX, aY, aZ spat3d ain, kX, kY, kZ, idist, ift, imode, imdel, iovr [, istor]

aW, aX, aY, aZ spat3di ain, iX, iY, iZ, idist, ift, imode [, istor]

spat3dt ioutft, iX, iY, iZ, idist, ift, imode, irlen [, iftnocl]

k1 spdist ifn, ktime, kx, ky

a1, a2, a3, a4 spsend

ar1[, ar2...] vbap asig, kazim [,
        kelev] [, kspread] [, ilayout]
array[] vbap asig, kazim [,
        kelev] [, kspread] [, ilayout]

ar1, ..., ar16 vbap16 asig, kazim [, kelev] [, kspread]

ar1, ..., ar16 vbap16move asig, idur, ispread, ifldnum, ifld1 \
        [, ifld2] [...]

ar1, ar2, ar3, ar4 vbap4 asig, kazim [, kelev] [, kspread]

ar1, ar2, ar3, ar4 vbap4move asig, idur, ispread, ifldnum, ifld1 \
        [, ifld2] [...]

ar1, ..., ar8 vbap8 asig, kazim [, kelev] [, kspread]

ar1, ..., ar8 vbap8move asig, idur, ispread, ifldnum, ifld1 \
        [, ifld2] [...]

k1[, k2...] vbapg kazim [,kelev] [, kspread] [, ilayout]
karray[] vbapg kazim [,kelev] [, kspread] [, ilayout]

kr1[, kr2...] vbapgmmove idur, ispread, ifldnum, ifld1 \
        [, ifld2] [...]
karray[] vbapgmmove idur, ispread, ifldnum, ifld1 \

```

```

    [, ifld2] [...]

vbaplsinit idim, ilsnum [, idir1] [, idir2] [...] [, idir32]
vbaplsinit idim, ilsnum, ilsarray

ar1[, ar2...] vbapmove asig, idur, ispread, ifldnum, ifld1 \
    [, ifld2] [...]
aarray[] vbapmove asig, idur, ispread, ifldnum, ifld1 \
    [, ifld2] [...]

vbapz inumchnls, istartndx, asig, kazim [, kelev] [, kspread]

vbapzmove inumchnls, istartndx, asig, idur, ispread, ifldnum, ifld1, \
    ifld2, [...]

```

### Signal Modifiers:Reverberation.

```

ares alpass asig, xrvt, ilpt [, iskip] [, insmps]

a1, a2 babo asig, ksrcx, ksrcy, ksrcz, irx, iry, irz [, idiff] [, ifno]

ares comb asig, krvt, ilpt [, iskip] [, insmps]

ares combinv asig, krvt, ilpt [, iskip] [, insmps]

aoutL, aoutR freeverb ainL, ainR, kRoomSize, kHFDamp[, iSRate[, iSkip]]

ares nestedap asig, imode, imaxdel, idel1, igain1 [, idel2] [, igain2] \
    [, idel3] [, igain3] [, istor]

ares nreverb asig, ktime, khdif [, iskip] [,inumCombs] [, ifnCombs] \
    [, inumAlpas] [, ifnAlpas]

a1[, a2, ...] platerev itabexcite. itabouts, kbndry, iaspect, istiff, idecay, iloss,
    aexcitel[, aexcite2, ...]

ares reverb asig, krvt [, iskip]

ares reverb2 asig, ktime, khdif [, iskip] [,inumCombs] \
    [, ifnCombs] [, inumAlpas] [, ifnAlpas]

aoutL, aoutR reverb3 ainL, ainR, kfblvl, kfco[, israte[, ipitchm[, iskip]]]

ares valpass asig, krvt, xlpt, imaxlpt [, iskip] [, insmps]

ares vcomb asig, krvt, xlpt, imaxlpt [, iskip] [, insmps]

```

### Signal Modifiers:Sample Level Operators.

```

denorm a1[, a2[, a3[, ... ]]]

ares diff asig [, iskip]
kres diff ksig [, iskip]

kres downsamp asig [, iwlen]

ares fold asig, kincr

ares integ asig [, iskip]

```

```

kres integ ksig [, iskip]

ares interp ksig [, iskip] [, imode]
    [, ivalue]

ares ntrpol asig1, asig2, kpoint [, imin] [, imax]
ires ntrpol isig1, isig2, ipoint [, imin] [, imax]
kres ntrpol ksig1, ksig2, kpoint [, imin] [, imax]

a(x) (control-rate args only)

i(x) (control-rate or init-rate arg)
i(karray,index1, ...) (k-array with indices)

k(x) (i-rate args only)
k(x) (a-rate args only)

S(x) (control-rate or init-rate arg)

ares samphold asig, agate [, ival] [, ivstor]
kres samphold ksig, kgate [, ival] [, ivstor]

ares upsamp ksig

kval valet kndx, avar

vaset kval, kndx, avar

```

### Signal Modifiers:Signal Limiters.

```

ares limit asig, klow, khigh
ires limit isig, ilow, ihigh
kres limit ksig, klow, khigh
ires[] limit isig[], ilow, ihigh
kres[] limit ksig[], klow, khigh

ares mirror asig, klow, khigh
ires mirror isig, ilow, ihigh
kres mirror ksig, klow, khigh

ares wrap asig, klow, khigh
ires wrap isig, ilow, ihigh
kres wrap ksig, klow, khigh

```

### Signal Modifiers:Special Effects.

```

ar distort asig, kdist, ifn[, ihp, istor]

ares distort1 asig, kpregain, kpostgain, kshapel, kshape2[, imode]

ares flanger asig, adel, kfeedback [, imaxd]

ares harmon asig, kestfrq, kmaxvar, kgenfreq1, kgenfreq2, imode, \
    iminfrq, iprd

ares harmon2 asig, koct, kfrq1, kfrq2, icpsmode, ilowest[, ipolarity]
ares harmon3 asig, koct, kfrq1, \
    kfrq2, kfrq3, icpsmode, ilowest[, ipolarity]
ares harmon4 asig, koct, kfrq1, \
    kfrq2, kfrq3, kfrq4, icpsmode, ilowest[, ipolarity]

```

ares **phaser1** asig, kfreq, kord, kfeedback [, iskip]  
 ares **phaser2** asig, kfreq, kq, kord, kmode, ksep, kfeedback

**Signal Modifiers:Standard Filters.**

ares **atone** asig, khp [, iskip]  
 ares **atonex** asig, khp [, inumlayer] [, iskip]  
 ares **atonex** asig, ahp [, inumlayer] [, iskip]  
 ares **biquad** asig, kb0, kb1, kb2, ka0, ka1, ka2 [, iskip]  
 ares **biquada** asig, ab0, ab1, ab2, aa0, aa1, aa2 [, iskip]  
 ares **butbp** asig, kfreq, kband [, iskip]  
 ares **butbr** asig, kfreq, kband [, iskip]  
 ares **buthp** asig, kfreq [, iskip]  
 ares **buthp** asig, afreq [, iskip]  
 ares **butlp** asig, kfreq [, iskip]  
 ares **butlp** asig, afreq [, iskip]  
 ares **butterbp** asig, xfreq, xband [, iskip]  
 ares **butterbr** asig, xfreq, xband [, iskip]  
 ares **butterhp** asig, kfreq [, iskip]  
 ares **butterhp** asig, afreq [, iskip]  
 ares **butterlp** asig, kfreq [, iskip]  
 ares **butterlp** asig, afreq [, iskip]  
 ares **clfilt** asig, kfreq, itype, inpol [, ikind] [, ipbr] [, isba] [, iskip]  
 asig **diode\_ladder** ain, xcf, xk [, inlp, isaturation, istor]  
 ashifted **doppler** asource, ksourceposition, kmicposition [, isoundspeed, ifiltercutoff]  
 asig **K35\_hpf** ain, xcf, xQ [, inlp, isaturation, istor]  
 asig **K35\_lpf** ain, xcf, xQ [, inlp, isaturation, istor]  
 ares **median** asig, ksize, imaxsize [, iskip]  
 kres **mediank** kin, ksize, imaxsize [, iskip]  
 aout **mode** ain, xfreq, xQ [, iskip]  
 ares **tone** asig, khp [, iskip]  
 ares **tonex** asig, khp [, inumlayer] [, iskip]  
 ares **tonex** asig, ahp [, inumlayer] [, iskip]  
 asig **zdf\_1pole** ain, xcf [, kmode, istor]  
 alp, ahp **zdf\_1pole\_mode** ain, xcf [, istor]  
 asig **zdf\_2pole** ain, xcf, xQ [, kmode, istor]

alp, abp, ahp **zdf\_2pole\_mode** ain, xcf, Q [, istor]  
 asig **zdf\_ladder** ain, xcf, xQ [, istor]

**Signal Modifiers:Standard Filters:Resonant.**

ares **areson** asig, kcf, kbw [, iscl] [, iskip]  
 ares **areson** asig, acf, kbw [, iscl] [, iskip]  
 ares **areson** asig, kcf, abw [, iscl] [, iskip]  
 ares **areson** asig, acf, abw [, iscl] [, iskip]  
  
 ares **bqrez** asig, xfco, xres [, imode] [, iskip]  
  
 ares **lowpass2** asig, kcf, kq [, iskip]  
  
 ares **lowres** asig, kcutoff, kresonance [, iskip]  
  
 ares **lowresx** asig, xcutoff, xresonance [, inumlayer] [, iskip]  
  
 ares **lpf18** asig, xfco, xres, xdist [, iskip]  
  
 asig **moogladder** ain, kcf, kres[, istor]  
 asig **moogladder** ain, acf, kres[, istor]  
 asig **moogladder** ain, kcf, ares[, istor]  
 asig **moogladder** ain, acf, ares[, istor]  
  
 asig **moogladder2** ain, kcf, kres[, istor]  
 asig **moogladder2** ain, acf, kres[, istor]  
 asig **moogladder2** ain, kcf, ares[, istor]  
 asig **moogladder2** ain, acf, ares[, istor]  
  
 ares **moogvcf** asig, xfco, xres [,iscale, iskip]  
  
 ares **moogvcf2** asig, xfco, xres [,iscale, iskip]  
  
 asig **mvchpf** ain, xcf[, istor]  
  
 asig **mvclpf1** ain, xcf, xres[,istor]  
  
 asig **mvclpf2** ain, xcf, xres[, istor]  
  
 asig **mvclpf3** ain, xcf, xres[, istor]  
  
 asig1,asig2,asig3,asig4 **mvclpf4** ain, xcf, xres[, istor]  
  
 ares **reson** asig, xcf, xbw [, iscl] [, iskip]  
  
 ares **resonr** asig, xcf, xbw [, iscl] [, iskip]  
  
 ares **resonx** asig, xcf, xbw [, inumlayer] [, iscl] [, iskip]  
  
 ares **resony** asig, kbf, kbw, inum, ksep [, isepmode] [, iscl] [, iskip]  
  
 ares **resonz** asig, xcf, xbw [, iscl] [, iskip]  
  
 ares **rezzy** asig, xfco, xres [, imode, iskip]  
  
 ahp,alp,abp,abr **statevar** ain, xcf, xq [, iosamps, istor]  
  
 alow, ahigh, aband **svfilter** asig, kcf, kq [, iscl] [, iskip]  
  
 ares **tbvcf** asig, xfco, xres, kdist, kasym [, iskip]

ares **vlowres** asig, kfco, kres, iord, ksep

### Signal Modifiers:Standard Filters:Control.

kres **aresonk** ksig, kcf, kbw [, iscl] [, iskip]

kres **atonek** ksig, khp [, iskip]

kres **lineto** ksig, ktime

kres **port** ksig, ihtim [, isig]

kres **portk** ksig, khtim [, isig]

kres **resonk** ksig, kcf, kbw [, iscl] [, iskip]

kres **resonxk** ksig, kcf, kbw[, inumlayer, iscl, istor]

aout **sc\_lag** ain, klagtime [, initialvalue=0]

kout **sc\_lag** kin, klagtime [, initialvalue=0]

aout **sc\_lagud** ain, klagup, klagdown

kout **sc\_lagud** kin, klagup, klagdown

aout **sc\_trig** ain, kdur

kout **sc\_trig** kin, kdur

kres **tlineto** ksig, ktime, ktrig

kres **tonek** ksig, khp [, iskip]

### Signal Modifiers:Specialized Filters.

ares **dcblock** ain [, igain]

ares **dcblock2** ain [, iorder] [, iskip]

asig **eqfil** ain, kcf, kbw, kgain[, istor]

ares **filter2** asig, iM, iN, ib0, ib1, ..., ibM, ia1, ia2, ..., iaN

kres **filter2** ksig, iM, iN, ib0, ib1, ..., ibM, ia1, ia2, ..., iaN

am, af **fmanal** are, aim

asig **fofilter** ain, xcf, xris, xdec[, istor]

ar1, ar2 **hilbert** asig

ar1, ar2 **hilbert2** asig, ifftsize, ihopsize

ares **nlfilt** ain, ka, kb, kd, kC, kL

ares **nlfilt2** ain, ka, kb, kd, kC, kL

ares **pareq** asig, kc, kv, kq [, imode] [, iskip]

ar **rbjeq** asig, kfco, klvl, kQ, kS[, imode]

ares **zfilter2** asig, kdamp, kfreq, iM, iN, ib0, ib1, ..., ibM, \

ia1,ia2, ..., iaN

### Signal Modifiers:Waveguides.

ares **wguide1** asig, xfreq, kcutoff, kfeedback

ares **wguide2** asig, xfreq1, xfreq2, kcutoff1, kcutoff2, \  
kfeedback1, kfeedback2

### Signal Modifiers:Waveshaping.

aout **chebyshevpoly** ain, k0 [, k1 [, k2 [...]]]

aout **pdclip** ain, kWidth, kCenter [, ibipolar [, ifullscale]]

aout **pdhalf** ain, kShapeAmount [, ibipolar [, ifullscale]]

aout **pdhalfy** ain, kShapeAmount [, ibipolar [, ifullscale]]

aout **powershape** ain, kShapeAmount [, ifullscale]

### Signal Modifiers:Comparators and Accumulators.

aout **cmp** aL, S\_operator, aR

amax **max** ain1, ain2 [, ain3] [, ain4] [...]

kmax **max** kin1, kin2 [, kin3] [, kin4] [...]

imax **max** iin1, iin2 [, iin3] [, iin4] [...]

knumkout **max\_k** asig, ktrig, itype

amax **maxabs** ain1, ain2 [, ain3] [, ain4] [...]

kmax **maxabs** kin1, kin2 [, kin3] [, kin4] [...]

**maxabsaccum** aAccumulator, aInput

**maxaccum** aAccumulator, aInput

amin **min** ain1, ain2 [, ain3] [, ain4] [...]

kmin **min** kin1, kin2 [, kin3] [, kin4] [...]

imin **min** iin1, iin2 [, iin3] [, iin4] [...]

amin **minabs** ain1, ain2 [, ain3] [, ain4] [...]

kmin **minabs** kin1, kin2 [, kin3] [, kin4] [...]

**minabsaccum** aAccumulator, aInput

**minaccum** aAccumulator, aInput

### Instrument Control:Clock Control.

**clockoff** inum

`clockon` inum

### Instrument Control:Conditional Values.

`(a == b ? v1 : v2)`

`(a >= b ? v1 : v2)`

`(a > b ? v1 : v2)`

`(a <= b ? v1 : v2)`

`(a < b ? v1 : v2)`

`(a != b ? v1 : v2)`

### Instrument Control:Compilation.

ires `compilecsd` Sfilename

ires `compileorc` Sfilename

ires `compilestr` Sorch

ires `evalstr` Scode

kres `evalstr` Scode, ktrig

`return` ival

### Instrument Control:Duration Control.

`ihold`

`turnoff`

`turnoff` inst

`turnoff` knst

`turnoff2` kinsno, kmode, krelease

`turnon` insnum [, itime]

### Instrument Control:Invocation.

`event` "scorechar", kinsnum, kdelay, kdur, [, kp4] [, kp5] [, ...]

`event` "scorechar", "insname", kdelay, kdur, [, kp4] [, kp5] [, ...]

`event_i` "scorechar", iinsnum, idelay, idur, [, ip4] [, ip5] [, ...]

`event_i` "scorechar", "insname", idelay, idur, [, ip4] [, ip5] [, ...]

`mute` insnum [, iswitch]

`mute` "insname" [, iswitch]

```

iHandle nstance insnum, iwhen, idur [, ip4] [, ip5] [...]
iHandle nstance "insname", iwhen, idur [, ip4] [, ip5] [...]

readscore Sin

remove insnum

schedkwhen ktrigger, kmintim, kmaxnum, kinsnum, kwhen, kdur \
[, ip4] [, ip5] [...]
schedkwhen ktrigger, kmintim, kmaxnum, "insname", kwhen, kdur \
[, ip4] [, ip5] [...]

schedkwhennamed ktrigger, kmintim, kmaxnum, "name", kwhen, kdur \
[, ip4] [, ip5] [...]

schedule insnum, iwhen, idur [, ip4] [, ip5] [...]
schedule "insname", iwhen, idur [, ip4] [, ip5] [...]

schedwhen ktrigger, kinsnum, kwhen, kdur [, ip4] [, ip5] [...]
schedwhen ktrigger, "insname", kwhen, kdur [, ip4] [, ip5] [...]

scoreline Sin, ktrig

scoreline_i Sin

```

### Instrument Control:Program Flow Control.

```

cggoto condition, label

cigoto condition, label

ckgoto condition, label

cngoto condition, label

else

elseif xa R xb then

endif

goto label

if ia R ib igoto label
if ka R kb kgoto label
if xa R xb goto label
if xa R xb then

igoto label

kgoto label

loop_ge indx, idecr, imin, label
loop_ge kndx, kdecr, kmin, label

loop_gt indx, idecr, imin, label
loop_gt kndx, kdecr, kmin, label

loop_le indx, incr, imax, label
loop_le kndx, kncr, kmax, label

loop_lt indx, incr, imax, label
loop_lt kndx, kncr, kmax, label

```

```

tigoto label

timeout istr, idur, label

until condition do
    ... od

while condition do
    ... od

```

### Instrument Control:Realtime Performance Control.

```

ir active insnum [,iopt [,inorel]]
ir active Sinsname [,iopt [,inorel]]
kres active kinsnum [,iopt [,inorel]]

ktot [,kcpu1, kcpu2,...]cpumeter ifreq

cpuprc insnum, ipercent
cpuprc Sinsname, ipercent

exitnow [ivalue]

jacktransport icommand [, ilocation]

maxalloc insnum, icount
maxalloc Sinsname, icount

prealloc insnum, icount
prealloc "insname", icount

```

### Instrument Control:Initialization and Reinitialization.

```

ares = xarg
ires = iarg
kres = karg
ires, ... = iarg, ...
kres, ... = karg, ...
table [ kval] = karg

ares init iarg
ires init iarg
kres init iarg
ares, ... init iarg, ...
ires, ... init iarg, ...
kres, ... init iarg, ...
tab init isize[, ival]

insno nstrnum "name"

p(x)

    ivar1, ... passign [istart]

pset icon1 [, icon2] [...]

reinit label

rigoto label

```

`rireturn`

`ir tival`

### Instrument Control:Sensing and Control.

`kres button` knum

`ktrig changed` kvar1 [, kvar2,..., kvarN]

`ktrig changed2` kvar1 [, kvar2,..., kvarN]

`ktrig changed2` karr[]

`ktrig changed2` aarr[]

`kres checkbox` knum

`kres control` knum

`ares follow` asig, idt

`ares follow2` asig, katt, krel

`Svalue getcfg` iopt

`kres joystick` kdevice ktab

`ktrig metro` kfreq [, initphase]

`ksig midifilestatus`

`ksig miditempo`

#### `p5gconnect`

`kres p5gdata` kcontrol

`icount pcount`

`kres peak` asig

`kres peak` ksig

`ivalue pindex` ipfieldIndex

`koct, kamp pitch` asig, iupdte, ilo, ihi, idbthresh [, ifrqs] [, iconf] \  
 [, istrtr] [, iocts] [, iq] [, inptls] [, irolloff] [, iskip]

`kcps, krms pitchamdf` asig, imincps, imaxcps [, icps] [, imedi] \  
 [, idowns] [, iexcps] [, irmsmedi]

`acps, alock plltrack` asig, kd [, kloopf, kloopq, klf, khf, kthresh]

`kcps, kamp ptrack` asig, ihopsize[,ipeaks]

`ival readscratch`[index]

#### `rewindscore`

`kres rms` asig [, ihp] [, iskip]

`kres[, kkeydown]` `sensekey`

`ktrig_out seqtime` ktime\_unit, kstart, kloop, kinitndx, kfn\_times

ktrig\_out **seqtime2** ktrig\_in, ktime\_unit, kstart, kloop, kinitndx, kfn\_times  
**setctrl** inum, ival, itype  
**setscorepos** ipos  
**splitrig** ktrig, kndx, imaxtics, ifn, kout1 [,kout2,...,koutN]  
 ktemp **tempest** kin, iprd, imindur, imemdur, ihp, ithresh, ihtim, ixfdbak, \  
     istartempo, ifn [, idisprd] [, itweek]  
**tempo** ktempo, istartempo  
 kres **tempoval**  
 ktrig **timedseq** ktempnt, ifn, kp1 [,kp2, kp3, ...,kpN]  
 kout **trigger** ksig, kthreshold, kmode  
**trigseq** ktrig\_in, kstart, kloop, kinitndx, kfn\_values, kout1 [, kout2] [...]  
 ares **vactrol** asig [iup, idown]  
 ires **wiiconnect** [itimeout, imaxnum]  
 kres **wiidata** kcontrol[, knum]  
**wiirange** icontrol, iminimum, imaximum[, inum]  
 kres **wiisend** kcontrol, kvalue[, knum]  
**writescratch**ival[, index]  
 kx, ky **xyin** iprd, ixmin, ixmax, iymin, iymax [, ixinit] [, iyinit]

### Instrument Control:Stacks.

xval1, [xval2, ... , xval31] **pop**  
 ival1, [ival2, ... , ival31] **pop**  
**fsig** **pop\_f**  
**push** xval1, [xval2, ... , xval31]  
**push** ival1, [ival2, ... , ival31]  
**push\_f** fsig  
**stack** iStackSize

### Instrument Control:Subinstrument Control.

a1, [...] [, a8] **subinstr** instrnum [, p4] [, p5] [...]  
 a1, [...] [, a8] **subinstr** "insname" [, p4] [, p5] [...]  
**subinstrinit** instrnum [, p4] [, p5] [...]  
**subinstrinit** "insname" [, p4] [, p5] [...]

### Instrument Control:Time Reading.

```

ir[, inano] date
kr[, knano] date

Sir dates [ itime]

ir readclock inum

ires rtclock
kres rtclock

kres timeinstk

kres timeinsts

ires timek
kres timek

ires times
kres times

```

### Jacko Opcodes.

```

asignal JackoAudioIn ScsoundPortName

JackoAudioInConnect SexternalPortName, ScsoundPortName

JackoAudioOut ScsoundPortName, asignal

JackoAudioOutConnect ScsoundPortName, SexternalPortName

JackoFreewheel [ienabled]

JackoInfo

JackoInit ServerName, SclientName

JackoMidiInConnect SexternalPortName, ScsoundPortName

JackoMidiOut ScsoundPortName, kstatus, kchannel, kdata1[, kdata2]

JackoMidiOutConnect ScsoundPortName, SexternalPortName

JackoNoteOut ScsoundPortName, kstatus, kchannel, kdata1[, kdata2]

JackoOn [iactive]

JackoTransport kcommand, [kposition]

```

### Lua Opcodes.

```

lua_exec Sluacode

lua_iopcall Sname, ...
lua_ikopcall Sname, ...
lua_iaopcall Sname, ...
lua_iopcall_off Sname, ...
lua_ikopcall_off Sname, ...

```

`lua_iaopcall_off` Sname, ...

`lua_opdef` Sname, Sluacode

### Serial I/O.

iPort `serialBegin` SPortName [, ibaudRate]

`serialEnd` iPort

`serialFlush` iPort

`serialPrint` iPort

kByte `serialRead` iPort

`serialWrite` iPort, iByte

`serialWrite` iPort, kByte

`serialWrite` iPort, SBytes

`serialWrite_i` iPort, iByte

`serialWrite_i` iPort, SBytes

### Table Control.

`ftfree` ifno, iwhen

gir `ftgen` ifn, itime, isize, igen, iarga [, iargb ] [...]

gir `ftgen` ifn, itime, isize, igen, iarray

ifno `ftgentmp` ip1, ip2dummy, isize, igen, iarga, iargb, ...

Sdst `getftargs` iftno, ktrig

`sndload` Sfname[, ifmt[, ichns[, isr[, ibas[, iamp[, istr \

[, ilpmod[, ilps[, ilpe]]]]]]]]]

### Table Control:Table Queries.

karray[] `array` ival1, ival2,....ivaln

karray[] `fillarray` ival1, ival2,....ivaln

karray `fillarray` ival1, ival2,....ivaln

`ftchnls`(x) (init-rate args only)

`ftcps`(x) (init-rate args only)

`ftlen`(x) (init-rate args only)

`ftlptim`(x) (init-rate args only)

`ftsr`(x) (init-rate args only)

karray `genarray` kstart, kens[, inc]

iarray `genarray` istart, iens[, inc]

```

karray genarray_i istart, iend [,inc]

ir lenarray karray[, iwhich]
kr lenarray karray[, iwhich]

karray maparray kinarray, String
karray maparray_i kinarray, String

nsamp(x) (init-rate args only)

karray slicearray kinarray, istart, iend

ires tbleng ifn
kres tbleng kfn

kr tabsum ifn[[, kmin] [, kmax]]

tb0_init ifn
tb1_init ifn
tb2_init ifn
tb3_init ifn
tb4_init ifn
tb5_init ifn
tb6_init ifn
tb7_init ifn
tb8_init ifn
tb9_init ifn
tb10_init ifn
tb11_init ifn
tb12_init ifn
tb13_init ifn
tb14_init ifn
tb15_init ifn
iout = tb0(iIndex)
kout = tb0(kIndex)
iout = tb1(iIndex)
kout = tb1(kIndex)
iout = tb2(iIndex)
kout = tb2(kIndex)
iout = tb3(iIndex)
kout = tb3(kIndex)
iout = tb4(iIndex)
kout = tb4(kIndex)
iout = tb5(iIndex)
kout = tb5(kIndex)
iout = tb6(iIndex)
kout = tb6(kIndex)
iout = tb7(iIndex)
kout = tb7(kIndex)
iout = tb8(iIndex)
kout = tb8(kIndex)
iout = tb9(iIndex)
kout = tb9(kIndex)
iout = tb10(iIndex)
kout = tb10(kIndex)
iout = tb11(iIndex)
kout = tb11(kIndex)
iout = tb12(iIndex)
kout = tb12(kIndex)
iout = tb13(iIndex)
kout = tb13(kIndex)
iout = tb14(iIndex)
kout = tb14(kIndex)
iout = tb15(iIndex)
kout = tb15(kIndex)

```

**Table Control:Dynamic Selection.**

```
ares tableikt xndx, kfn [, ixmode] [, ixoff] [, iwrap]
kres tableikt kndx, kfn [, ixmode] [, ixoff] [, iwrap]

ares tablekt xndx, kfn [, ixmode] [, ixoff] [, iwrap]
kres tablekt kndx, kfn [, ixmode] [, ixoff] [, iwrap]

ares tablexkt xndx, kfn, kwrap, iwsiz [ , ixmode] [, ixoff] [, iwrap]
```

**Table Control:Read/Write Opreations.**

```
ftload "filename", iflag, ifn1 [, ifn2] [...]

ftloadk "filename", ktrig, iflag, ifn1 [, ifn2] [...]

iNumberOfFile ftsamplebank SDirectory, iFirstTableNumber, iSkipTime, iFormat, iChannel,
kNumberOfFile ftsamplebank SDirectory, kFirstTableNumber, kTrigger, kSkipTime, kFormat,
kChannel,

ftsave "filename", iflag, ifn1 [, ifn2] [...]

ftsavek "filename", ktrig, iflag, ifn1 [, ifn2] [...]

ptablew asig, andx, ifn [, ixmode] [, ixoff] [, iwgmde]
ptablew isig, indx, ifn [, ixmode] [, ixoff] [, iwgmde]
ptablew ksig, kndx, ifn [, ixmode] [, ixoff] [, iwgmde]

tablecopy kdft, ksft

knumpassed tablefilter kouttable, kintatble, kmode, kparam

inumpassed tablefilteri iouttable, iintatble, imode, iparam

tablegpw kfn

tableicopy idft, isft

tableigpw ifn

tableimix idft, idoff, ilen, islft, isloff, islg, is2ft, is2off, is2g

tableiw isig, indx, ifn [, ixmode] [, ixoff] [, iwgmde]

tablemix kdft, kdoff, klen, ks1ft, ksloff, ks1g, ks2ft, ks2off, ks2g

ares tablera kfn, kstart, koff

tablew asig, andx, ifn [, ixmode] [, ixoff] [, iwgmde]
tablew isig, indx, ifn [, ixmode] [, ixoff] [, iwgmde]
tablew ksig, kndx, ifn [, ixmode] [, ixoff] [, iwgmde]

kstart tablewa kfn, asig, koff

tablewkt asig, andx, kfn [, ixmode] [, ixoff] [, iwgmde]
tablewkt ksig, kndx, kfn [, ixmode] [, ixoff] [, iwgmde]

kout tabmorph kindex, kweightpoint, ktabnum1, ktabnum2, \
ifn1, ifn2 [, ifn3, ifn4, ...,ifnN]

aout tabmorpha aindex, aweightpoint, atabnum1, atabnum2, \
```

```

    ifn1, ifn2 [, ifn3, ifn4, ... ifnN]
aout tabmorphak aindex, kweightpoint, ktabnum1, ktabnum2, \
    ifn1, ifn2 [, ifn3, ifn4, ... ifnN]
kout tabmorphi kindex, kweightpoint, ktabnum1, ktabnum2, \
    ifn1, ifn2 [, ifn3, ifn4, ..., ifnN]
tabplay ktrig, knumtics, kfn, kout1 [,kout2,..., koutN]
tabrec ktrig_start, ktrig_stop, knumtics, kfn, kin1 [,kin2,...,kinN]

```

### FLTK:Containers.

```

FLgroup "label", iwidth, iheight, ix, iy [, iborder] [, image]
FLgroupEnd
FLpack iwidth, iheight, ix, iy, itype, ispace, iborder
FLpackEnd
FLpanel "label", iwidth, iheight [, ix] [, iy] [, iborder] [, ikbdcapture] [, iclose]
FLpanelEnd
FLscroll iwidth, iheight [, ix] [, iy]
FLscrollEnd
FLtabs iwidth, iheight, ix, iy
FLtabsEnd

```

### FLTK:Valuators.

```

kout, ihandle FLcount "label", imin, imax, istep1, istep2, itype, \
    iwidth, iheight, ix, iy, iopcode [, kp1] [, kp2] [, kp3] [...] [, kpN]
koutx, kouty, ihandlex, ihandley FLjoy "label", iminx, imaxx, iminy, \
    imaxy, iexp, iexpy, idisp, idispy, iwidth, iheight, ix, iy
kout, ihandle FLknob "label", imin, imax, iexp, itype, idisp, iwidth, \
    ix, iy [, icursorsize]
kout, ihandle FLroller "label", imin, imax, istep, iexp, itype, idisp, \
    iwidth, iheight, ix, iy
kout, ihandle FLslider "label", imin, imax, iexp, itype, idisp, iwidth, \
    iheight, ix, iy
kout, ihandle FLtext "label", imin, imax, istep, itype, iwidth, \
    iheight, ix, iy

```

### FLTK:Other.

```

ihandle FLbox "label", itype, ifont, isize, iwidth, iheight, ix, iy [, image]

kout, ihandle FLbutBank itype, inumx, inumy, iwidth, iheight, ix, iy, \
    iopcode [, kp1] [, kp2] [, kp3] [, kp4] [, kp5] [...] [, kpN]

kout, ihandle FLbutton "label", ion, ioff, itype, iwidth, iheight, ix, \
    iy, iopcode [, kp1] [, kp2] [, kp3] [, kp4] [, kp5] [...] [, kpN]

ihandle FLcloseButton "label", iwidth, iheight, ix, iy

ihandle FLexecButton "command", iwidth, iheight, ix, iy

inumsnap FLgetsnap index [, igroup]

ihandle FLhvsBox inumlinesX, inumlinesY, iwidth, iheight, ix, iy

FLhvsBox kx, ky, ihandle

kascii FLkeyIn [ifn]

FLloadsnap "filename" [, igroup]

kx, ky, kb1, kb2, kb3 FLmouse [imode]

FLprintk itime, kval, idisp

FLprintk2 kval, idisp

FLrun

FLsavesnap "filename" [, igroup]

inumsnap, inumval FLsetsnap index [, ifn, igroup]

FLsetSnapGroup igroup

FLsetVal ktrig, kvalue, ihandle

FLsetVal_i ivalue, ihandle

FLslidBnk "names", inumsliders [, ioutable] [, iwidth] [, iheight] [, ix] \
    [, iy] [, itypetable] [, iexptable] [, istart_index] [, iminmaxtable]

FLslidBnk2 "names", inumsliders, ioutable, iconfigtable [,iwidth, iheight, ix, iy, is-
    tart_index]
FLslidBnk2 istring, inumsliders, ioutable, iconfigtable [,iwidth, iheight, ix, iy, is-
    tart_index]

FLslidBnk2Set ihandle, ifn [, istartIndex, istartSlid, inumSlid]

FLslidBnk2Setk ktrig, ihandle, ifn [, istartIndex, istartSlid, inumSlid]

ihandle FLslidBnkGetHandle

FLslidBnkSet ihandle, ifn [, istartIndex, istartSlid, inumSlid]

FLslidBnkSetk ktrig, ihandle, ifn [, istartIndex, istartSlid, inumSlid]

FLupdate

ihandle FLvalue "label", iwidth, iheight, ix, iy

FLvkeybd "keyboard.map", iwidth, iheight, ix, iy

FLvslidBnk "names", inumsliders [, ioutable] [, iwidth] [, iheight] [, ix] \
    [, iy] [, itypetable] [, iexptable] [, istart_index] [, iminmaxtable]

```

**FLvslidBnk2** "names", inumsliders, ioutable, iconfigtable [,iwidth, iheight, ix, iy, istart\_index]

koutx, kouty, kinside **FLxyin** ioutx\_min, ioutx\_max, iouty\_min, iouty\_max, \  
iwindx\_min, iwindx\_max, iwindy\_min, iwindy\_max [, iexpx, iexpy, ioutx, iouty]

**vphaseseg** kphase, ioutab, ielems, itab1, idist1, itab2 \  
[, idist2, itab3, ... , idistN-1, itabN]

### **FLTK:Appearance.**

**FLcolor** ired, igreen, iblue [, ired2, igreen2, iblue2]

**FLcolor2** ired, igreen, iblue

**FLhide** ihandle

**FLlabel** isize, ifont, ialign, ired, igreen, iblue

**FLsetAlign** ialign, ihandle

**FLsetBox** itype, ihandle

**FLsetColor** ired, igreen, iblue, ihandle

**FLsetColor2** ired, igreen, iblue, ihandle

**FLsetFont** ifont, ihandle

**FLsetPosition** ix, iy, ihandle

**FLsetSize** iwidth, iheight, ihandle

**FLsetText** "itext", ihandle

**FLsetTextColor** ired, iblue, igreen, ihandle

**FLsetTextSize** isize, ihandle

**FLsetTextType** itype, ihandle

**FLshow** ihandle

### **Mathematical Operations:Arithmetic and Logic Operations.**

a + b (no rate restriction)

a / b (no rate restriction)

a % b (no rate restriction)

a \* b (no rate restriction)

a && b (logical AND; not audio-rate)

a & b (bitwise AND)

~ a (bitwise NOT)

`a | b` (bitwise OR)  
`a << b` (bitshift left)  
`a >> b` (bitshift left)  
`a # b` (bitwise NON EQUIVALENCE)  
`a || b` (logical OR; not audio-rate)  
`a ^ b` (b not audio-rate)  
`a # b` (no rate restriction)

**Mathematical Operations:Comparators and Accumulators.**

`clear` avar1 [, avar2] [, avar3] [...]  
`vincr` accum, aincr

**Mathematical Operations:Mathematical Functions.**

`abs(x)` (no rate restriction)  
`abs(k/i[])` (k- or i-arrays )  
`ceil(x)` (init-, control-, or audio-rate arg allowed)  
`ceil(k/i[])` (k- or i-arrays )  
`exp(x)` (no rate restriction)  
`exp(k/i[])` (k- or i-arrays )  
`floor(x)` (init-, control-, or audio-rate arg allowed)  
`floor(k/i[])` (k- or i-arrays )  
`frac(x)` (init-rate or control-rate args; also works at audio rate in Csound5)  
`frac(k/i[])` (k- or i-arrays )  
`int(x)` (init-rate or control-rate; also works at audio rate in Csound5)  
`log10(x)` (no rate restriction)  
`log10(k/i[])` (k- or i-arrays )  
`log2(x)` (no rate restriction)  
`log2(k/i[])` (k- or i-arrays )  
`logbtwo(x)` (init-rate or control-rate args only)  
`powoftwo(x)` (init-rate or control-rate args only)  
`qinf(x)` (no rate restriction)  
`qnan(x)` (no rate restriction)  
`round(x)` (init-, control-, or audio-rate arg allowed)  
`round(k/i[])` (k- or i-arrays )  
`sqrt(x)` (no rate restriction)

`sqrt(k/i[])` (k- or i-arrays )

### Mathematical Operations:Trigonometric Functions.

`cos(x)` (no rate  
restriction)

`cos(k/i[])` (k- or i-arrays )

`cosh(x)` (no rate  
restriction)

`cosh(k/i[])` (k- or i-arrays )

`cosinv(x)` (no rate  
restriction)

`cosinv(k/i[])` (k- or i-arrays )

`signum(x)` (no rate restriction)

`sin(x)` (no rate restriction)

`sin(k/i[])` (k- or i-arrays )

`sinh(x)` (no rate  
restriction)

`sinh(k/i[])` (k- or i-arrays )

`sininv(x)` (no rate  
restriction)

`sininv(k/i[])` (k- or i-arrays)

`tan(x)` (no rate  
restriction)

`tan(k/i[])` (k- or i-arrays )

`tanh(x)` (no rate  
restriction)

`tanh(k/i[])` (k- or i-arrays )

`taninv(x)` (no rate  
restriction)

`taninv(k/i[])` (k- or i-arrays )

### Mathematical Operations:Amplitude Functions.

`ampdb(x)` (no rate restriction)

`ampdbfs(x)` (no rate restriction)

`db(x)`

`dbamp(x)` (init-rate or control-rate args only)

`dbfsamp(x)` (init-rate or control-rate args only)

### Mathematical Operations:Random Functions.

`birnd(x)` (init- or control-rate only)

`rnd(x)` (init- or control-rate only)

### Mathematical Operations:Opcode Equivalents of Functions.

```
ares divz xa, xb, ksubst
ires divz ia, ib, isubst
kres divz ka, kb, ksubst
...divz(ka, kb, ksubst)... (no rate restriction)

ares mac ksig1, asig1 [, ksig2] [, asig2] [, ksig3] [, asig3] [...]
ares maca asig1 , asig2 [, asig3] [, asig4] [, asig5] [...]

aout polynomial ain, k0 [, k1 [, k2 [...]]]

ares pow aarg, kpow [, inorm]
ires pow iarg, ipow [, inorm]
kres pow karg, kpow [,
    inorm]
ires[] pow iarg[], ipow[]
kres[] pow karg[], kpow[]
ires[] pow iarg[], ipow
kres[] pow karg[], kpow

ares product asig1, asig2 [, asig3] [...]

ares sum asig1 [, asig2] [, asig3] [...]
kres sum karr
ires sum iarr

ares taninv2 ay, ax
ires taninv2 iy, ix
kres taninv2 ky, kx
...taninv2(ky, kx)... (no rate restriction)
```

### Pitch Converters:Functions.

```
cent(x)

cpsmidinn (MidiNoteNumber) (init- or control-rate args only)

cpsoct (oct) (no rate restriction)

cpspch (pch) (init- or control-rate args only)

imidi ftom ifreq
kmidi ftom kfreq

ifreq mtof imidi
kfreq mtof kmidi

Snote mton kmidi
Snote mton imidi

kmidi ntom Snote
imidi ntom Snote

octave(x)

octcps (cps) (init- or control-rate args only)
```

**octmidinn** (MidiNoteNumber) (init- or control-rate args only)

**octpch** (pch) (init- or control-rate args only)

**pchmidinn** (MidiNoteNumber) (init- or control-rate args only)

**pchoct** (oct) (init- or control-rate args only)

imidi **pchtom** ipch  
 kmidi **pchtom** kpch

**semitone**(x)

**Pitch Converters:Tuning Opcodes.**

icps **cps2pch** ipch, iequal

kcps **cpstun** ktrig, kindex, kfn

icps **cpstuni** index, ifn

icps **cpsxpch** ipch, iequal, irepeat, ibase

**Real-time MIDI:Input.**

kaft **aftouch** [imin] [, imax]

ival **chanctrl** ichnl, ictlno [, ilow] [, ihigh]  
 kval **chanctrl** ichnl, ictlno [, ilow] [, ihigh]

idest **ctrl14** ichan, ictlno1, ictlno2, imin, imax [, ifn]  
 kdest **ctrl14** ichan, ictlno1, ictlno2, kmin, kmax [, ifn]

idest **ctrl21** ichan, ictlno1, ictlno2, ictlno3, imin, imax [, ifn]  
 kdest **ctrl21** ichan, ictlno1, ictlno2, ictlno3, kmin, kmax [, ifn]

idest **ctrl7** ichan, ictlno, imin, imax [, ifn]  
 kdest **ctrl7** ichan, ictlno, kmin, kmax [, ifn]  
 adest **ctrl7** ichan, ictlno, kmin, kmax [, ifn] [, icutoff]

**ctrlinit** ichnl, ictlno1, ival1 [, ictlno2] [, ival2] [, ictlno3] \  
 [, ival3] [,...ival32]

**initc14** ichan, ictlno1, ictlno2, ivalue

**initc21** ichan, ictlno1, ictlno2, ictlno3, ivalue

**initc7** ichan, ictlno, ivalue

**massign** ichnl, insnum[, ireset]  
**massign** ichnl, "insname"[, ireset]

idest **midic14** ictlno1, ictlno2, imin, imax [, ifn]  
 kdest **midic14** ictlno1, ictlno2, kmin, kmax [, ifn]

idest **midic21** ictlno1, ictlno2, ictlno3, imin, imax [, ifn]  
 kdest **midic21** ictlno1, ictlno2, ictlno3, kmin, kmax [, ifn]

idest **midic7** ictlno, imin, imax [, ifn]

```

kdest midic7 ictlno, kmin, kmax [, ifn]

ival midictrl inum [, imin] [, imax]
kval midictrl inum [, imin] [, imax]

ival notnum

ibend pchbend [imin] [, imax]
kbend pchbend [imin] [, imax]

pgmassign ipgm, inst[, ichn]
pgmassign ipgm, "insname"[, ichn]

ires polyaft inote [, ilow] [, ihigh]
kres polyaft inote [, ilow] [, ihigh]

ival veloc [ilow] [, ihigh]

```

### Real-time MIDI:Output.

```

nrrpn kchan, kparmnum, kparmvalue

outiat ichn, ivalue, imin, imax

outic ichn, inum, ivalue, imin, imax

outic14 ichn, imsb, ilsb, ivalue, imin, imax

outipat ichn, inotenum, ivalue, imin, imax

outipb ichn, ivalue, imin, imax

outipc ichn, iprog, imin, imax

outkat kchn, kvalue, kmin, kmax

outkc kchn, knum, kvalue, kmin, kmax

outkc14 kchn, kmsb, klsb, kvalue, kmin, kmax

outkpat kchn, knotenum, kvalue, kmin, kmax

outkpb kchn, kvalue, kmin, kmax

outkpc kchn, kprog, kmin, kmax

```

### Real-time MIDI:Converters.

```

iamp ampmidi iscal [, ifn]

iamplitude ampmidid ivelocity, idecibels
kamplitude ampmidid kvelocity, idecibels

icps cpsmidi

icps cpsmidib [irange]
kcps cpsmidib [irange]

icps cpstmid ifn

```

ioct [octmidi](#)

ioct [octmidib](#) [irange]  
 koct [octmidib](#) [irange]

ipch [pchmidi](#)

ipch [pchmidib](#) [irange]  
 kpch [pchmidib](#) [irange]

### Real-time MIDI:Generic I/O.

kstatus, kchan, kdata1, kdata2 [midiin](#)

[midiout](#) kstatus, kchan, kdata1, kdata2

### Real-time MIDI:Event Extenders.

kflag [release](#)

[xtratim](#) iextradur

### Real-time MIDI>Note Output.

[midion](#) kchn, knum, kvel

[midion2](#) kchn, knum, kvel, ktrig

[moscil](#) kchn, knum, kvel, kdur, kpause

[noteoff](#) ichn, inum, ivel

[noteon](#) ichn, inum, ivel

[noteondur](#) ichn, inum, ivel, idur

[noteondur2](#) ichn, inum, ivel, idur

### Real-time MIDI:MIDI/Score Interoperability.

[midichannelaftertouch](#) xchannelaftertouch [, ilow] [, ihigh]

ichn [midichn](#)

[midicontrolchange](#) xcontroller, xcontrollervalue [, ilow] [, ihigh]

[mididefault](#) xdefault, xvalue

[midinoteoff](#) xkey, xvelocity

[midinoteoncps](#) xcps, xvelocity

[midinoteonkey](#) xkey, xvelocity

**midinoteonoct** xoct, xvelocity  
**midinoteonpch** xpch, xvelocity  
**midipitchbend** xpitchbend [, ilow] [, ihigh]  
**midipolyaftertouch** xpolyaftertouch, xcontrollervalue [, ilow] [, ihigh]  
**midiprogramchange** xprogram

### Real-time MIDI:System Realtime.

**mclock** ifreq  
**mrtmsg** imsgtype

### Real-time MIDI:Slider Banks.

i1,...,i16 **s16b14** ichan, ictrlno\_msb1, ictrlno\_lsb1, imin1, imax1, \  
 initvalue1, ifn1,..., ictrlno\_msb16, ictrlno\_lsb16, imin16, imax16, initvalue16,  
 ifn16  
 k1,...,k16 **s16b14** ichan, ictrlno\_msb1, ictrlno\_lsb1, imin1, imax1, \  
 initvalue1, ifn1,..., ictrlno\_msb16, ictrlno\_lsb16, imin16, imax16, initvalue16,  
 ifn16  
 i1,...,i32 **s32b14** ichan, ictrlno\_msb1, ictrlno\_lsb1, imin1, imax1, \  
 initvalue1, ifn1,..., ictrlno\_msb32, ictrlno\_lsb32, imin32, imax32, initvalue32,  
 ifn32  
 k1,...,k32 **s32b14** ichan, ictrlno\_msb1, ictrlno\_lsb1, imin1, imax1, \  
 initvalue1, ifn1,..., ictrlno\_msb32, ictrlno\_lsb32, imin32, imax32, initvalue32,  
 ifn32  
 i1,...,i16 **slider16** ichan, ictrlnum1, imin1, imax1, init1, ifn1,..., \  
 ictrlnum16, imin16, imax16, init16, ifn16  
 k1,...,k16 **slider16** ichan, ictrlnum1, imin1, imax1, init1, ifn1,..., \  
 ictrlnum16, imin16, imax16, init16, ifn16  
 k1,...,k16 **slider16f** ichan, ictrlnum1, imin1, imax1, init1, ifn1, \  
 icutoff1,..., ictrlnum16, imin16, imax16, init16, ifn16, icutoff16  
 kflag **slider16table** ichan, ioutTable, ioffset, ictrlnum1, imin1, imax1, \  
 init1, ifn1, .... , ictrlnum16, imin16, imax16, init16, ifn16  
 kflag **slider16tablef** ichan, ioutTable, ioffset, ictrlnum1, imin1, imax1, \  
 init1, ifn1, icutoff1, .... , ictrlnum16, imin16, imax16, init16, ifn16, icutoff16  
 i1,...,i32 **slider32** ichan, ictrlnum1, imin1, imax1, init1, ifn1,..., \  
 ictrlnum32, imin32, imax32, init32, ifn32  
 k1,...,k32 **slider32** ichan, ictrlnum1, imin1, imax1, init1, ifn1,..., \  
 ictrlnum32, imin32, imax32, init32, ifn32  
 k1,...,k32 **slider32f** ichan, ictrlnum1, imin1, imax1, init1, ifn1, icutoff1, \  
 ..., ictrlnum32, imin32, imax32, init32, ifn32, icutoff32  
 kflag **slider32table** ichan, ioutTable, ioffset, ictrlnum1, imin1, \  
 imax1, init1, ifn1, .... , ictrlnum32, imin32, imax32, init32, ifn32  
 kflag **slider32tablef** ichan, ioutTable, ioffset, ictrlnum1, imin1, imax1, \  
 init1, ifn1, icutoff1, .... , ictrlnum32, imin32, imax32, init32, ifn32, icutoff32

```

i1,...,i64 slider64 ichan, ictlnum1, imin1, imax1, init1, ifn1,..., \
    ictlnum64, imin64, imax64, init64, ifn64
k1,...,k64 slider64 ichan, ictlnum1, imin1, imax1, init1, ifn1,..., \
    ictlnum64, imin64, imax64, init64, ifn64

k1,...,k64 slider64f ichan, ictlnum1, imin1, imax1, init1, ifn1, \
    icutoff1,..., ictlnum64, imin64, imax64, init64, ifn64, icutoff64

kflag slider64table ichan, ioutTable, ioffset, ictlnum1, imin1, \
    imax1, init1, ifn1, .... , ictlnum64, imin64, imax64, init64, ifn64

kflag slider64tablef ichan, ioutTable, ioffset, ictlnum1, imin1, imax1, \
    init1, ifn1, icutoff1, .... , ictlnum64, imin64, imax64, init64, ifn64, icutoff64

i1,...,i8 slider8 ichan, ictlnum1, imin1, imax1, init1, ifn1,..., \
    ictlnum8, imin8, imax8, init8, ifn8
k1,...,k8 slider8 ichan, ictlnum1, imin1, imax1, init1, ifn1,..., \
    ictlnum8, imin8, imax8, init8, ifn8

k1,...,k8 slider8f ichan, ictlnum1, imin1, imax1, init1, ifn1, icutoff1, \
    ..., ictlnum8, imin8, imax8, init8, ifn8, icutoff8

kflag slider8table ichan, ioutTable, ioffset, ictlnum1, imin1, imax1, \
    init1, ifn1,..., ictlnum8, imin8, imax8, init8, ifn8

kflag slider8tablef ichan, ioutTable, ioffset, ictlnum1, imin1, imax1, \
    init1, ifn1, icutoff1, .... , ictlnum8, imin8, imax8, init8, ifn8, icutoff8

k1, k2, ...., k16 sliderKawai imin1, imax1, init1, ifn1, \
    imin2, imax2, init2, ifn2, ..., imin16, imax16, init16, ifn16

```

### Signal Flow Graph Opcodes.

```

alwayson Tinstrument [p4, ..., pn]

connect Tsource1, Soutlet1, Tsink1, Sinlet1

ifno ftgenonce ip1, ip2dummy, isize, igen, iarga, iargb, ...

asignal inleta Sname

fsignal inletf Sname

ksignal inletk Sname

ksignal inletkid Sname, SinstanceID

array inletv Sname

outleta Sname, asignal

outletf Sname, fsignal

outletk Sname, ksignal

outletkid Sname, SinstanceID, ksignal

outletv Sname, array

```

### Spectral Processing:STFT.

```

htableseg ifn1, idur1, ifn2 [, idur2] [, ifn3] [...]
ares pvadd ktmpnt, kfmmod, ifilcod, ifn, ibins [, ibinoffset] \
    [, ibinincr] [, iextractmode] [, ifreqlim] [, igatefn]

pvbufread ktmpnt, ifile

ares pvcross ktmpnt, kfmmod, ifile, kampscale1, kampscale2 [, ispecwp]

ares pvinterp ktmpnt, kfmmod, ifile, kfreqscale1, kfreqscale2, \
    kampscale1, kampscale2, kfreqinterp, kampinterp

ares pvoc ktmpnt, kfmmod, ifilcod [, ispecwp] [, iextractmode] \
    [, ifreqlim] [, igatefn]

kfreq, kamp pvread ktmpnt, ifile, ibin

tableseg ifn1, idur1, ifn2 [, idur2] [, ifn3] [...]

tablexseg ifn1, idur1, ifn2 [, idur2] [, ifn3] [...]

ares vpvoc ktmpnt, kfmmod, ifile [, ispecwp] [, ifn]

```

### Spectral Processing:LPC.

```

ares lpfreson asig, kfrqratio

lpinterp islot1, islot2, kmix

krmsr, krmso, kerr, kcps lpread ktmpnt, ifilcod [, inpoles] [, ifrmrate]

ares lpreson asig

lpslot islot

```

### Spectral Processing:Non-Standard.

```

wsig specaddm wsig1, wsig2 [, imul2]

wsig specdiff wsigin

specdisp wsig, iprd [, iwtflg]

wsig specfilt wsigin, ifhtim

wsig spechist wsigin

koct, kamp specptrk wsig, kvar, ilo, ihi, istr, idbthresh, inptls, \
    irolloff [, ioddd] [, iconfs] [, interp] [, ifprd] [, iwtflg]

wsig specscal wsigin, ifscale, ifthresh

ksum specsum wsig [, interp]

wsig spectrum xsig, iprd, iocts, ifrqa [, iq] [, ihann] [, idbout] \
    [, idsprd] [, idsinrs]

```

**Spectral Processing:Streaming.**

```

fsig binit fin, isize

fsig codanal ain, ifftsize, ioverlap, iwinsize, iwintype [, iformat] [, iinit]

asig codaslicing ain, amod, iwinsize

asig codasynth kamp, kfreq, itab, iftab, iatab[, inum]
asig codasynth fsig, kamp, kfreq[, inum]
asig codasynth fsig

ftrks partials ffr, fphs, kthresh, kminpts, kmaxgap, imaxtracks

kframe pvs2tab tvar|kvar[], fsig
kframe pvs2tab kmags[], kfreqs[], fsig

ares pvsadsyn fsrc, inoscs, kfmmod [, ibinoffset] [, ibinincr] [, iinit]

fsig pvsanal ain, ifftsize, ioverlap, iwinsize, iwintype [, iformat] [, iinit]

fsig pvsarp fsigin, kbin, kdepth, kgain

fsig pvsbandp fsigin, xlowcut, xlowfull, \
    xhighfull, xhighcut[, ktype]

fsig pvsbandr fsigin, xlowcut, xlowfull, \
    xhighfull, xhighcut[, ktype]

kamp, kfr pvsbin fsig, kbin

fsig pvsblur fsigin, kblurtime, imaxdel

ihandle, ktime pvsbuffer fsig, ilen

fsig pvsbufread ktime, khandle[, ilo, ihi, iclear]

fsig pvsbufread2 ktime, khandle, ift1, ift2

fsig pvscale fsigin, kscal[, kkeepform, kgain, kcoefs]

kcent pvscent fsig

keps[] pvsceps fsig[, icoefs]

fsig pvsconnect fsrc, fdest, kamp1, kamp2

fsig pvsdemix fleft, fright, kpos, kwidth, ipoints

fsig pvsdiskin SFname,ktscal,kgain[,ioffset, ichan]

pvsdisp fsig[, ibins, iwtflg]

fsig pvsfilter fsigin, fsigfil, kdepth[, igain]

fsig pvsfread ktimpt, ifn [, ichan]

fsig pvsfreeze fsigin, kfreeza, kfreezf

pvsftr fsrc, ifna [, ifnf]

kflag pvsftw fsrc, ifna [, ifnf]

pvsfwrite fsig, ifile

fsig pvsgain fsigin, kgain

```

fsig **pvshift** fsigin, kshift, klowest[, kkeepform, igain, kcoefs]  
 ffr,fphs **pvsifd** ain, ifftsize, ihopsize, iwintype[,iscal]  
 fsig **pvsin** kchan[, isize, iolap, iwinsize, iwintype, iformat]  
 ioverlap, inumbins, iwinsize, iformat **pvsinfo** fsrc  
 fsig **pvsinit** isize[, iolap, iwinsize, iwintype, iformat]  
 fsig **pvslock** fsigin, klock  
 fsig **pvsmaska** fsrc, ifn, kdepth  
 fsig **pvsmix** fsigin1, fsigin2  
 fsig **pvssmooth** fsigin, kacf, kcf  
 fsig **pvsmorph** fsig1, fsig2, kampint, kfrqint  
 fsig **pvsosc** kamp, kfreq, ktype, isize [,ioverlap] [, iwinsize] [, iwintype] [, iformat]  
**pvsout** fsig, kchan  
 kfr, kamp **pvspitch** fsig, kthresh  
 fsig **pvstanal** ktimescal, kamp, kpitch, ktab, [kdetect, kwrap, ioffset,ifftsize, ihop,  
 idbthresh]  
  
 fsig **pvstencil** fsigin, kgain, klevel, iftable  
 fsig **pvstrace** fsigin, kn  
 fsig **pvsvoc** famp, fexc, kdepth, kgain [,kcoefs]  
 fsig **pvs warp** fsigin, kscal, kshift[, klowest, kmeth, kgain, kcoefs]  
 ares **pvsynth** fsrc, [iinit]  
 asig **resyn** fin, kscal, kpitch, kmaxtracks, ifn  
 asig **sinsyn** fin, kscal, kmaxtracks, ifn  
  
 fsig **tab2pvs** tvar|karr[[,ihopsize, iwinsize, iwintype]  
 fsig **tab2pvs** kmags[], kfregs[[,ihopsize, iwinsize, iwintype]  
 ffr,fphs **tabifd** ktimpt, kamp, kpitch, ifftsize, ihopsize, iwintype,ifn  
 asig **tradsyn** fin, kscal, kpitch, kmaxtracks, ifn  
 fsig **trcross** fin1, fin2, ksearch, kdepth [, kmode]  
 fsig **trfilter** fin, kamnt, ifn  
 fsig, kfr, kamp **trhighest** fin1, kscal  
 fsig, kfr, kamp **trlowest** fin1, kscal  
 fsig **trmix** fin1, fin2  
 fsig **trscale** fin, kpitch[, kgain]  
 fsig **trshift** fin, kpshift[, kgain]  
 fsiglow, fsighi **trsplit** fin, ksplit[, kgainlow, kgainhigh]

### Spectral Processing:ATS.

```

ar ATSadd ktimepnt, kfmmod, iatsfile, ifn, ipartialsl, ipartialoffset, \
    ipartialincr, igatefn]

ar ATSaddnz ktimepnt, iatsfile, ibands[, ibandoffset, ibandincr]

ATSbufread ktimepnt, kfmmod, iatsfile, ipartialsl, ipartialoffset, \
    ipartialincr]

ar ATScross ktimepnt, kfmmod, iatsfile, ifn, kmylev, kbuflev, ipartialsl \
    [, ipartialoffset, ipartialincr]

idata ATSinfo iatsfile, ilocation

kamp ATSinterpread kfreq

kfrq, kamp ATSpartialtap ipartialnum

kfreq, kamp ATSread ktimepnt, iatsfile, ipartial

kenegy ATSreadnz ktimepnt, iatsfile, iband

ar ATSSinnoi ktimepnt, ksinlev, knzlev, kfmmod, iatsfile, ipartialsl \
    [, ipartialoffset, ipartialincr]

```

### Spectral Processing:Loris.

```

lorismorph isrcidx, itgtidx, istoreidx, kfreqmorphenv, kampmorphenv, kbwmorphenv

ar lorisplay ireadidx, kfreqenv, kampenv, kbwenv

lorisread ktimepnt, ifilcod, istoreidx, kfreqenv, kampenv, kbwenv[, ifadetime]

```

### Spectral Processing:Other.

```

kcent centroid asig, ktrig, ifftsize

asig[,asig2] filesca1 ktimesca1, kamp, kpitch, Sfile, klock [,ifftsize, idecim,
    ithresh]

asig mincer atimpt, kamp, kpitch, ktab, klock[,ifftsize,idecim]

asig, asig2, ktime mp3sca1 Sfile,ktimesca1,kpitch, kamp [,iskip,ifftsize, idecim,
    ilock]

asig paulstretch istretch, iwindowsize, ift

asig temposca1 ktimesca1, kamp, kpitch, ktab, klock [,ifftsize, idecim, ithresh]

```

### Strings:Definition.

Sdst `strfromurl` StringURL

Sdst `strget` indx

`strset` iarg, istring

### Strings:Manipulation.

`puts` Sstr, ktrig[, inonl]

Sdst `sprintf` Sfmt, xarg1[, xarg2[, ... ]]

Sdst `sprintfk` Sfmt, xarg1[, xarg2[, ... ]]

Sdst `strcat` Ssrc1, Ssrc2

Sdst `strcatk` Ssrc1, Ssrc2

ires `strcmp` S1, S2

kres `strcmpk` S1, S2

Sdst `strcpy` Ssrc

Sdst = Ssrc

Sdst `strcpyk` Ssrc

ipos `strindex` S1, S2

kpos `strindexk` S1, S2

ilen `strlen` Sstr

klen `strlenk` Sstr

ipos `strrindex` S1, S2

kpos `strrindexk` S1, S2

Sdst `strsub` Ssrc[, istart[, iend]]

Sdst `strsubk` Ssrc, kstart, kend

### Strings:Conversion.

ichr `strchar` Sstr[, ipos]

kchr `strchark` Sstr[, kpos]

Sdst `strlower` Ssrc

Sdst `strlowerk` Ssrc

ir `strtod` Sstr

ir `strtod` indx

```
kr strtodk Sstr
kr strtodk kndx

ir strtol Sstr
ir strtol indx

kr strtolk Sstr
kr strtolk kndx

Sdst strupper Ssrc

Sdst strupperk Ssrc
```

### Vectorial:Tables.

```
vtaba andx, ifn, aout1 [, aout2, aout3, .... , aoutN ]
vtabi indx, ifn, iout1 [, iout2, iout3, .... , ioutN ]
vtabk kndx, ifn, kout1 [, kout2, kout3, .... , koutN ]
vtablelk kfn,kout1 [, kout2, kout3, .... , koutN ]
vtablea andx, kfn, kinterp, ixmode, aout1 [, aout2, aout3, .... , aoutN ]
vtablei indx, ifn, interp, ixmode, iout1 [, iout2, iout3, .... , ioutN ]
vtablek kndx, kfn, kinterp, ixmode, kout1 [, kout2, kout3, .... , koutN ]
vtablewa andx, kfn, ixmode, ainarg1 [, ainarg2, ainarg3 , .... , ainargN ]
vtablewi indx, ifn, ixmode, inarg1 [, inarg2, inarg3 , .... , inargN ]
vtablewk kndx, kfn, ixmode, kinarg1 [, kinarg2, kinarg3 , .... , kinargN ]
vtabwa andx, ifn, ainarg1 [, ainarg2, ainarg3 , .... , ainargN ]
vtabwi indx, ifn, inarg1 [, inarg2, inarg3 , .... , inargN ]
vtabwk kndx, ifn, kinarg1 [, kinarg2, kinarg3 , .... , kinargN ]
```

### Vectorial:Scalar operations.

```
vadd ifn, kval, kelements [, kdstoffset] [, kverbose]
vadd_i ifn, ival, ielements [, idstoffset]
vexp ifn, kval, kelements [, kdstoffset] [, kverbose]
vexp_i ifn, ival, ielements[, idstoffset]
vmult ifn, kval, kelements [, kdstoffset] [, kverbose]
vmult_i ifn, ival, ielements [, idstoffset]
vpow ifn, kval, kelements [, kdstoffset] [, kverbose]
vpow_i ifn, ival, ielements [, idstoffset]
```

**Vectorial:Vectorial operations.**

**vaddv** ifn1, ifn2, kelements [, kdstoffset] [, ksrcoffset] [,kverbose]  
**vaddv\_i** ifn1, ifn2, ielements [, idstoffset] [, isrcoffset]  
**vcopy** ifn1, ifn2, kelements [, kdstoffset] [, ksrcoffset] [, kverbose]  
**vcopy\_i** ifn1, ifn2, ielements [,idstoffset, isrcoffset]  
**vdivv** ifn1, ifn2, kelements [, kdstoffset] [, ksrcoffset] [,kverbose]  
**vdivv\_i** ifn1, ifn2, ielements [, idstoffset] [, isrcoffset]  
**vexpv** ifn1, ifn2, kelements [, kdstoffset] [, ksrcoffset] [,kverbose]  
**vexpv\_i** ifn1, ifn2, ielements [, idstoffset] [, isrcoffset]  
**vmap** ifn1, ifn2, ielements [,idstoffset, isrcoffset]  
**vmultv** ifn1, ifn2, kelements [, kdstoffset] [, ksrcoffset] [,kverbose]  
**vmultv\_i** ifn1, ifn2, ielements [, idstoffset] [, isrcoffset]  
**vpowv** ifn1, ifn2, kelements [, kdstoffset] [, ksrcoffset] [,kverbose]  
**vpowv\_i** ifn1, ifn2, ielements [, idstoffset] [, isrcoffset]  
**vsubv** ifn1, ifn2, kelements [, kdstoffset] [, ksrcoffset] [,kverbose]  
**vsubv\_i** ifn1, ifn2, ielements [, idstoffset] [, isrcoffset]

**Vectorial:Envelopes.**

**vexpseg** ifnout, ielements, ifn1, idur1, ifn2 [, idur2, ifn3 [...]]  
**vlinseg** ifnout, ielements, ifn1, idur1, ifn2 [, idur2, ifn3 [...]]

**Vectorial:Limiting and Wrapping.**

**vlimit** ifn, kmin, kmax, ielements  
**vmirror** ifn, kmin, kmax, ielements  
**vwrap** ifn, kmin, kmax, ielements

**Vectorial:Delay Paths.**

kout **vdelayk** ksig, kdel, imaxdel [, iskip, imode]  
**vecdelay** ifn, ifnIn, ifnDel, ielements, imaxdel [, iskip]  
**vport** ifn, khtime, ielements [, ifnInit]

### Vectorial:Random.

**vrandh** ifn, krange, kcps, ielements [, idstoffset] [, iseed] \  
 [, isize] [, ioffset]

**vrandi** ifn, krange, kcps, ielements [, idstoffset] [, iseed] \  
 [, isize] [, ioffset]

### Vectorial:Cellular Automata.

**cell** ktrig, kreinit, ioutFunc, initStateFunc, iRuleFunc, ielements

**vcella** ktrig, kreinit, ioutFunc, initStateFunc, \  
 iRuleFunc, ielements, irulelen [, iradius]

### Zak Patch System.

**zacl** kfirst, klast

**zakinit** isizea, isizek

ares **zamod** asig, kzamod

ares **zar** kndx

ares **zarg** kndx, kgain

**zaw** asig, kndx

**zawm** asig, kndx [, imix]

ir **zir** indx

**ziw** isig, indx

**ziwm** isig, indx [, imix]

**zkcl** kfirst, klast

kres **zkmod** ksig, kzmod

kres **zkr** kndx

**zkw** ksig, kndx

**zkwm** ksig, kndx [, imix]

### Plugin Hosting:DSSI and LADSPA.

**dssiactivate** ihandle, ktoggle

[aout1, aout2, ..., aout9] **dssiaudio** ihandle, [ain1, ain2, ..., ain9]

**dssictls** ihandle, iport, kvalue, ktrigger

ihandle **dssiinit** ilibraryname, ipluginindex [, iverbose]

**dssilist**

### Plugin Hosting:VST.

aout1,aout2 **vstaudio** instance, [ain1, ain2]

aout1,aout2 **vstaudiog** instance, [ain1, ain2]

**vstbankload** instance, ipath

**vstedit** instance

**vstinfo** instance

instance **vstinit** ilibrarypath [,iverbose]

**vstmidiout** instance, kstatus, kchan, kdata1, kdata2

**vstnote** instance, kchan, knote, kveloc, kdur

**vstparamset** instance, kparam, kvalue

kvalue **vstparamget** instance, kparam

**vstprogset** instance, kprogram

### OSC.

ihandle **OSCinit** iport

kans **OSClisten** ihandle, idest, itype [, xdata1, xdata2, ...]

Smess[],klen **OSCraw** iport

**OSCsend** kwhen, ihost, iport, idestination, itype [, xdata1, xdata2, ...]

### Faust Opcodes.

ihandle,a1[,a2,...] **faustaudio** ifac[,ain1,...]

ihandle **faustcompile** Scode, Sargs

**faustctl** idsp,Scontrol,kval

ihandle,a1[,a2,...] **faustgen** SCode[,ain1,...]

### Network.

**remoteport** iportnum

asig **sockrecv** iport, ilength

ksig **sockrecv** iport, ilength

asigl, asigr **sockrecvs** iport, ilength  
String **sockrecv** iport, ilength  
asig **strecv** Sipaddr, iport

**socksend** asig, Sipaddr, iport, ilength  
**socksend** ksig, Sipaddr, iport, ilength  
**socksends** asigl, asigr, Sipaddr, iport,  
    ilength  
**stsend** asig, Sipaddr, iport

### Remote Opcodes.

**insglobal** isource, instrnum [,instrnum...]  
**insremot** idestination, isource, instrnum [,instrnum...]  
**midglobal** isource, instrnum [,instrnum...]  
**midremot** idestination, isource, instrnum [,instrnum...]

### Mixer Opcodes.

**MixerClear**  
kgain **MixerGetLevel** isend, ibuss  
asignal **MixerReceive** ibuss, ichannel  
**MixerSend** asignal, isend, ibuss, ichannel  
**MixerSetLevel** isend, ibuss, kgain  
**MixerSetLevel\_i** isend, ibuss, igain

### Ableton Link Opcodes.

**link\_beat\_force** i\_peer, k\_beat [, k\_at\_time\_seconds [, k\_quantum ]]  
k\_beat\_number, k\_phase, k\_current\_time\_seconds **link\_beat\_get** i\_peer [, k\_quantum]  
**link\_beat\_request** i\_peer, k\_beat [, k\_at\_time\_seconds [, k\_quantum ]]  
i\_peer **link\_create** [i\_bpm]  
**ableton\_link\_enable** i\_peer [, k\_enable]  
k\_is\_enabled **link\_is\_enabled** i\_peer  
k\_trigger, k\_beat, k\_phase, k\_current\_time\_seconds **link\_metro** i\_peer [, k\_quantum]  
k\_count **link\_peers** i\_peer  
k\_bpm **link\_tempo\_get** i\_peer  
**link\_tempo\_set** i\_peer, k\_bpm [, k\_at\_time\_seconds]

**Python Opcodes.**

```

pyassign "variable", kvalue
pyassigni "variable", ivalue
plassign "variable", kvalue
plassigni "variable", ivalue
pyassignt ktrigger, "variable", kvalue
plassignt ktrigger, "variable", kvalue

kresult
kresult1, kresult2
kr1, kr2, kr3
kr1, kr2, kr3, kr4
kr1, kr2, kr3, kr4, kr5
kr1, kr2, kr3, kr4, kr5, kr6
kr1, kr2, kr3, kr4, kr5, kr6, kr7
kr1, kr2, kr3, kr4, kr5, kr6, kr7, kr8

pycall "callable", karg1, ...
pycall1 "callable", karg1, ...
pycall2 "callable", karg1, ...
pycall3 "callable", karg1, ...
pycall4 "callable", karg1, ...
pycall5 "callable", karg1, ...
pycall6 "callable", karg1, ...
pycall7 "callable", karg1, ...
pycall8 "callable", karg1, ...
pycallt ktrigger, "callable", karg1, ...
pycall1t ktrigger, "callable", karg1, ...
pycall2t ktrigger, "callable", karg1, ...
pycall3t ktrigger, "callable", karg1, ...
pycall4t ktrigger, "callable", karg1, ...
pycall5t ktrigger, "callable", karg1, ...
pycall6t ktrigger, "callable", karg1, ...
pycall7t ktrigger, "callable", karg1, ...
pycall8t ktrigger, "callable", karg1, ...
pycalli "callable", karg1, ...
pycall1i "callable", iarg1, ...
pycall2i "callable", iarg1, ...
pycall3i "callable", iarg1, ...
pycall4i "callable", iarg1, ...
pycall5i "callable", iarg1, ...
pycall6i "callable", iarg1, ...
pycall7i "callable", iarg1, ...
pycall8i "callable", iarg1, ...
pycalln "callable", nresults, kresult1, ..., kresultn, karg1, ...
pycallni "callable", nresults, irect1, ..., irectn, iarg1, ...

kresult
kresult1, kresult2
kr1, kr2, kr3
kr1, kr2, kr3, kr4
kr1, kr2, kr3, kr4, kr5
kr1, kr2, kr3, kr4, kr5, kr6
kr1, kr2, kr3, kr4, kr5, kr6, kr7
kr1, kr2, kr3, kr4, kr5, kr6, kr7, kr8

pylcall "callable", karg1, ...
pylcall1 "callable", karg1, ...
pylcall2 "callable", karg1, ...
pylcall3 "callable", karg1, ...
pylcall4 "callable", karg1, ...
pylcall5 "callable", karg1, ...
pylcall6 "callable", karg1, ...
pylcall7 "callable", karg1, ...
pylcall8 "callable", karg1, ...
pylcallt ktrigger, "callable", karg1, ...
pylcall1t ktrigger, "callable", karg1, ...
pylcall2t ktrigger, "callable", karg1, ...
pylcall3t ktrigger, "callable", karg1, ...
pylcall4t ktrigger, "callable", karg1, ...
pylcall5t ktrigger, "callable", karg1, ...
pylcall6t ktrigger, "callable", karg1, ...
pylcall7t ktrigger, "callable", karg1, ...
pylcall8t ktrigger, "callable", karg1, ...
pylcalli "callable", karg1, ...
pylcall1i "callable", iarg1, ...
pylcall2i "callable", iarg1, ...
pylcall3i "callable", iarg1, ...
pylcall4i "callable", iarg1, ...
pylcall5i "callable", iarg1, ...
pylcall6i "callable", iarg1, ...
pylcall7i "callable", iarg1, ...
pylcall8i "callable", iarg1, ...

kresult
kresult1, kresult2
kr1, kr2, kr3
kr1, kr2, kr3, kr4
kr1, kr2, kr3, kr4, kr5
kr1, kr2, kr3, kr4, kr5, kr6
kr1, kr2, kr3, kr4, kr5, kr6, kr7
kr1, kr2, kr3, kr4, kr5, kr6, kr7, kr8

irect
irect1, irect2
ir1, ir2, ir3
ir1, ir2, ir3, ir4
ir1, ir2, ir3, ir4, ir5
ir1, ir2, ir3, ir4, ir5, ir6
ir1, ir2, ir3, ir4, ir5, ir6, ir7
ir1, ir2, ir3, ir4, ir5, ir6, ir7, ir8

```

`pycalln` "callable", nresults, kresult1, ..., kresultn, karg1, ...  
`pycallni` "callable", nresults, irestult1, ..., irestultn, iarg1, ...

kresult `pyeval` "expression"  
irestult `pyevali` "expression"  
kresult `pyleval` "expression"  
irestult `pylevali` "expression"  
kresult `pyevalt` ktrigger, "expression"  
kresult `pylevalt` ktrigger, "expression"

`pyexec` "filename"  
`pyexeci` "filename"  
`pylexec` "filename"  
`pylexeci` "filename"  
`pyexec` ktrigger, "filename"  
`plyexec` ktrigger, "filename"

#### `pyinit`

`pyrun` "statement"  
`pyruni` "statement"  
`pylrun` "statement"  
`pylruni` "statement"  
`pyrunt` ktrigger, "statement"  
`pylrunt` ktrigger, "statement"

### Image Processing Opcodes.

iimagenum `imagecreate` iwidth, iheight

`imagefree` iimagenum

ared, agreen, ablue `imagegetpixel` iimagenum, ax, ay  
kred, kgreen, kblue `imagegetpixel` iimagenum, kx, ky

iimagenum `imageload` filename

`imagesave` iimagenum, filename

`imagesetpixel` iimagenum, ax, ay, ared, agreen, ablue  
`imagesetpixel` iimagenum, kx, ky, kred, kgreen, kblue

iwidth, iheight `imagesize` iimagenum

### Array Opcodes.

`copya2ftab` kftbl, tab

`copyf2array` tab, kftbl

kmax [,kindx] `maxarray` karray

kmin [,kindx] `minarray` karray

`scalearray` tarray, kmin, kmax[, kleft, kright]

ksum `sumarray` karray

### Miscellaneous.

```

SFiles[] directory SDirectory[, SExtention]

kfl fareylen kfn

ifl fareyleni ifn

modmatrix iresfn, isrcmodfn, isrcparmf, imodscale, inum_mod, \\
inum_parm, kupdate

Sres pwd

aout select a1, a2, aless, aequal, amore

ires system_i itrig, Scmd, [inowait]
kres system ktrig, Scmd, [knowait]

tableshuffle ktablenum
tableshufflei itablenum

```

### Utilities.

```

csound -U atsa [flags] infilename outfilename

cs [-OPTIONS] <name> [CSOUND OPTIONS ... ]

csb64enc [OPTIONS ... ] infile1 [ infile2 [ ... ]]

csound -U cvanal [flags] infilename outfilename
cvanal [flags] infilename outfilename

dnoise [flags] -i noise_ref_file -o output_soundfile input_soundfile

envext [-flags] soundfile
csound -U envext [-flags] soundfile

extractor [OPTIONS ... ] infile

het_export het_file ctext_file
csound -U het_export het_file ctext_file

het_import ctext_file het_file
csound -U het_import ctext_file het_file

csound -U hetro [flags] infilename outfilename
hetro [flags] infilename outfilename

csound -U lpanal [flags] infilename outfilename
lpanal [flags] infilename outfilename

makecsd [OPTIONS ... ] infile1 [ infile2 [ ... ]]

mixer [OPTIONS ... ] infile [[OPTIONS... ] infile] ...

mkdb [-m] [base_directory]

pv_export pv_file ctext_file
csound -U pv_export pv_file ctext_file

pv_import ctext_file pv_file
csound -U pv_import ctext_file pv_file

```

**csound -U pvanal** [flags] infilename outfilename  
**pvanal** [flags] infilename outfilename

**csound -U pvlook** [flags] infilename  
**pvlook** [flags] infilename

**scale** [OPTIONS ... ] infile

**sdif2ad** [flags] infilename outfilename

**csound -U sndinfo** [options] soundfilenames ...  
**sndinfo** [options] soundfilenames ...

**srconv** [flags] infile

---

# Appendix A. List of examples

## **Orchestra Syntax:Header.**

*0dbfs.csd* [examples/0dbfs.csd]

*nchnls.csd* [examples/nchnls.csd]

*nchnls\_i.csd* [examples/nchnls\_i.csd]

*sr.csd* [examples/sr.csd]

## **Orchestra Syntax:Block Statements.**

*endin.csd* [examples/endin.csd]

*endop.csd* [examples/endop.csd]

*instr.csd* [examples/instr.csd]

*opcode\_example.csd* [examples/opcode\_example.csd]

## **Orchestra Syntax:Macros.**

*define.csd* [examples/define.csd]

*define\_args.csd* [examples/define\_args.csd]

*define.csd* [examples/define.csd]

*define\_args.csd* [examples/define\_args.csd]

*include.csd* [examples/include.csd]

## **Signal Generators:Additive Synthesis/Resynthesis.**

*adsyn.csd* [examples/adsyn.csd]

*adsynt.csd* [examples/adsynt.csd]

*adsynt2.csd* [examples/adsynt2.csd]

*hsboscil.csd* [examples/hsboscil.csd]

*hsboscil\_midi.csd* [examples/hsboscil\_midi.csd]

## **Signal Generators:Basic Oscillators.**

*lfo.csd* [examples/lfo.csd]

*oscblk.csd* [examples/oscblk.csd]

*oscil.csd* [examples/oscil.csd]

*oscil3.csd* [examples/oscil3.csd]

*oscili.csd* [examples/oscili.csd]

*oscilikt.csd* [examples/oscilikt.csd]

*osciliktp.csd* [examples/osciliktp.csd]

*oscilikts.csd* [examples/oscilikts.csd]

*osciln.csd* [examples/osciln.csd]

*oscils.csd* [examples/oscils.csd]

*poscil.csd* [examples/poscil.csd]

*poscil3.csd* [examples/poscil3.csd]

*poscil3-file.csd* [examples/poscil3-file.csd]

*vibr.csd* [examples/vibr.csd]

*vibrato.csd* [examples/vibrato.csd]

#### **Signal Generators:Dynamic Spectrum Oscillators.**

*buzz.csd* [examples/buzz.csd]

*gbuzz.csd* [examples/gbuzz.csd]

*mpulse.csd* [examples/mpulse.csd]

*vco.csd* [examples/vco.csd]

*vco2.csd* [examples/vco2.csd]

*vco2ift.csd* [examples/vco2ift.csd]

*vco2init.csd* [examples/vco2init.csd]

#### **Signal Generators:FM Synthesis.**

*crossfm.csd* [examples/crossfm.csd]

*fmb3.csd* [examples/fmb3.csd]

*fmbell.csd* [examples/fmbell.csd]

*fmmetal.csd* [examples/fmmetal.csd]

*fmpercfl.csd* [examples/fmpercfl.csd]

*fmrhode.csd* [examples/fmrhode.csd]

*fmvoice.csd* [examples/fmvoice.csd]

*fmwurlie.csd* [examples/fmwurlie.csd]

*foscil.csd* [examples/foscil.csd]

*foscili.csd* [examples/foscili.csd]

**Signal Generators:Granular Synthesis.**

*diskgrain.csd* [examples/diskgrain.csd]

*fof.csd* [examples/fof.csd]

*fof2.csd* [examples/fof2.csd]

*fof2-2.csd* [examples/fof2-2.csd]

*fog.csd* [examples/fog.csd]

*grain.csd* [examples/grain.csd]

*grain2.csd* [examples/grain2.csd]

*grain3.csd* [examples/grain3.csd]

*granule.csd* [examples/granule.csd]

*partikkel.csd* [examples/partikkel.csd]

*partikkel-2.csd* [examples/partikkel-2.csd]

*partikkel-panlaws.csd* [examples/partikkel-panlaws.csd]

*partikkelgetset.csd* [examples/partikkelgetset.csd]

*partikkelgetset.csd* [examples/partikkelgetset.csd]

*partikkelsync.csd* [examples/partikkelsync.csd]

*sndwarp.csd* [examples/sndwarp.csd]

*sndwarpst.csd* [examples/sndwarpst.csd]

*syncgrain.csd* [examples/syncgrain.csd]

*syncloop.csd* [examples/syncloop.csd]

*vosim.csd* [examples/vosim.csd]

**Signal Generators:Hyper Vectorial Synthesis.**

*hvs1.csd* [examples/hvs1.csd]

*hvs2.csd* [examples/hvs2.csd]

*hvs2-2.csd* [examples/hvs2-2.csd]

*hvs3.csd* [examples/hvs3.csd]

**Signal Generators:Linear and Exponential Generators.**

*bpf.csd* [examples/bpf.csd]  
*cosseg.csd* [examples/cosseg.csd]  
*cossegb.csd* [examples/cossegb.csd]  
*cossegr.csd* [examples/cossegr.csd]  
*expcurve.csd* [examples/expcurve.csd]  
*expon.csd* [examples/expon.csd]  
*expseg.csd* [examples/expseg.csd]  
*expsega.csd* [examples/expsega.csd]  
*expsegb.csd* [examples/expsegb.csd]  
*expsegba.csd* [examples/expsegba.csd]  
*expsegr.csd* [examples/expsegr.csd]  
*gainslider.csd* [examples/gainslider.csd]  
*line.csd* [examples/line.csd]  
*linlin.csd* [examples/linlin.csd]  
*linseg.csd* [examples/linseg.csd]  
*linsegb.csd* [examples/linsegb.csd]  
*linsegr.csd* [examples/linsegr.csd]  
*logcurve.csd* [examples/logcurve.csd]  
*loopseg.csd* [examples/loopseg.csd]  
*loopsegp.csd* [examples/loopsegp.csd]  
*looptseg.csd* [examples/looptseg.csd]  
*loopxseg.csd* [examples/loopxseg.csd]  
*lpshold.csd* [examples/lpshold.csd]  
*scale.csd* [examples/scale.csd]  
*transeg.csd* [examples/transeg.csd]  
*transegb.csd* [examples/transegb.csd]  
*transegr.csd* [examples/transegr.csd]  
*xyscale.csd* [examples/xyscale.csd]

**Signal Generators:Envelope Generators.**

*adsr.csd* [examples/adsr.csd]  
*envlpx.csd* [examples/envlpx.csd]  
*envlpxr.csd* [examples/envlpxr.csd]  
*linen.csd* [examples/linen.csd]  
*linenr.csd* [examples/linenr.csd]  
*madsr.csd* [examples/madsr.csd]  
*madsr-2.csd* [examples/madsr-2.csd]  
*mxadsr.csd* [examples/mxadsr.csd]  
*xadsr.csd* [examples/xadsr.csd]

**Signal Generators:Models and Emulations.**

*bamboo.csd* [examples/bamboo.csd]  
*barmodel.csd* [examples/barmodel.csd]  
*cabasa.csd* [examples/cabasa.csd]  
*chuap.csd* [examples/chuap.csd]  
*crunch.csd* [examples/crunch.csd]  
*dripwater.csd* [examples/dripwater.csd]  
*gendy.csd* [examples/gendy.csd]  
*gendy-2.csd* [examples/gendy-2.csd]  
*gendyc.csd* [examples/gendyc.csd]  
*gendyx.csd* [examples/gendyx.csd]  
*gendyx-2.csd* [examples/gendyx-2.csd]  
*gogobel.csd* [examples/gogobel.csd]  
*guiro.csd* [examples/guiro.csd]  
*lorenz.csd* [examples/lorenz.csd]  
*mandel.csd* [examples/mandel.csd]  
*mandol.csd* [examples/mandol.csd]  
*marimba.csd* [examples/marimba.csd]  
*moog.csd* [examples/moog.csd]  
*planet.csd* [examples/planet.csd]

*prepiano.csd* [examples/prepiano.csd]

*sandpaper.csd* [examples/sandpaper.csd]

*sekere.csd* [examples/sekere.csd]

*shaker.csd* [examples/shaker.csd]

*sleighbells.csd* [examples/sleighbells.csd]

*stix.csd* [examples/stix.csd]

*tambourine.csd* [examples/tambourine.csd]

*vibes.csd* [examples/vibes.csd]

*voice.csd* [examples/voice.csd]

**Signal Generators:Phasors.**

*phasor.csd* [examples/phasor.csd]

*phasorbnk.csd* [examples/phasorbnk.csd]

*sc\_phasor.csd* [examples/sc\_phasor.csd]

*syncphasor.csd* [examples/syncphasor.csd]

*syncphasor-CZresonance.csd* [examples/syncphasor-CZresonance.csd]

**Signal Generators:Random (Noise) Generators.**

*betarand.csd* [examples/betarand.csd]

*bexprnd.csd* [examples/bexprnd.csd]

*cauchy.csd* [examples/cauchy.csd]

*cauchy.csd* [examples/cauchy.csd]

*cuserrnd.csd* [examples/cuserrnd.csd]

*duserrnd.csd* [examples/duserrnd.csd]

*dust.csd* [examples/dust.csd]

*dust2.csd* [examples/dust2.csd]

*exprand.csd* [examples/exprand.csd]

*exprandi.csd* [examples/exprandi.csd]

*fractalnoise.csd* [examples/fractalnoise.csd]

*gauss.csd* [examples/gauss.csd]

*gaussi.csd* [examples/gaussi.csd]

*gausstrig.csd* [examples/gausstrig.csd]  
*gausstrig-2.csd* [examples/gausstrig-2.csd]  
*jitter.csd* [examples/jitter.csd]  
*jitter2.csd* [examples/jitter2.csd]  
*jspline.csd* [examples/jspline.csd]  
*linrand.csd* [examples/linrand.csd]  
*noise.csd* [examples/noise.csd]  
*noise-2.csd* [examples/noise-2.csd]  
*pcauchy.csd* [examples/pcauchy.csd]  
*pinker.csd* [examples/pinker.csd]  
*pinkish.csd* [examples/pinkish.csd]  
*poisson.csd* [examples/poisson.csd]  
*rand.csd* [examples/rand.csd]  
*randh.csd* [examples/randh.csd]  
*randi.csd* [examples/randi.csd]  
*random.csd* [examples/random.csd]  
*randomh.csd* [examples/randomh.csd]  
*randomi.csd* [examples/randomi.csd]  
*rnd31.csd* [examples/rnd31.csd]  
*rnd31\_krate.csd* [examples/rnd31\_krate.csd]  
*rnd31\_seed7.csd* [examples/rnd31\_seed7.csd]  
*rnd31\_time.csd* [examples/rnd31\_time.csd]  
*rspline.csd* [examples/rspline.csd]  
*seed.csd* [examples/seed.csd]  
*trandom.csd* [examples/trandom.csd]  
*trirand.csd* [examples/trirand.csd]  
*unirand.csd* [examples/unirand.csd]  
*urandom.csd* [examples/urandom.csd]  
*urandom\_krate.csd* [examples/urandom\_krate.csd]  
*urd.csd* [examples/urd.csd]

*weibull.csd* [examples/weibull.csd]

**Signal Generators:Sample Playback.**

*bbcutm.csd* [examples/bbcutm.csd]

*bbcuts.csd* [examples/bbcuts.csd]

*flooper.csd* [examples/flooper.csd]

*flooper2.csd* [examples/flooper2.csd]

*fluidAllOut.csd* [examples/fluidAllOut.csd]

*fluidCCi.csd* [examples/fluidCCi.csd]

*fluidCCK.csd* [examples/fluidCCK.csd]

*fluidcomplex.csd* [examples/fluidcomplex.csd]

*fluidEngine.csd* [examples/fluidEngine.csd]

*fluidLoad.csd* [examples/fluidLoad.csd]

*fluidNote.csd* [examples/fluidNote.csd]

*fluidOut.csd* [examples/fluidOut.csd]

*fluidProgramSelect.csd* [examples/fluidProgramSelect.csd]

*fluidcomplex.csd* [examples/fluidcomplex.csd]

*fluidSetInterpMethod.csd* [examples/fluidSetInterpMethod.csd]

*loscil.csd* [examples/loscil.csd]

*loscil3.csd* [examples/loscil3.csd]

*loscilx.csd* [examples/loscilx.csd]

*lphasor.csd* [examples/lphasor.csd]

*lposcil.csd* [examples/lposcil.csd]

*lposcil3.csd* [examples/lposcil3.csd]

*lposcila.csd* [examples/lposcila.csd]

*lposcilsa.csd* [examples/lposcilsa.csd]

*lposcilsa2.csd* [examples/lposcilsa2.csd]

*sfilist.csd* [examples/sfilist.csd]

*sfinstr.csd* [examples/sfinstr.csd]

*sfinstr3.csd* [examples/sfinstr3.csd]

*sfinstr3m.csd* [examples/sfinstr3m.csd]

*sfload.csd* [examples/sfload.csd]

*sflooper.csd* [examples/sflooper.csd]

*sfpassign.csd* [examples/sfpassign.csd]

*sfplay3.csd* [examples/sfplay3.csd]

*sfplay3m.csd* [examples/sfplay3m.csd]

*sfplaym.csd* [examples/sfplaym.csd]

*sfplist.csd* [examples/sfplist.csd]

*sfpreset.csd* [examples/sfpreset.csd]

*sndloop.csd* [examples/sndloop.csd]

*waveset.csd* [examples/waveset.csd]

#### **Signal Generators:Scanned Synthesis.**

*scans.csd* [examples/scans.csd]

*scans-2.csd* [examples/scans-2.csd]

*scantable.csd* [examples/scantable.csd]

*xscanmap.csd* [examples/xscanmap.csd]

*xscans.csd* [examples/xscans.csd]

*xscanu.csd* [examples/xscanu.csd]

#### **Signal Generators:STK Opcodes.**

*STKBandedWG.csd* [examples/STKBandedWG.csd]

*STKBeeThree.csd* [examples/STKBeeThree.csd]

*STKBlowBotl.csd* [examples/STKBlowBotl.csd]

*STKBlowHole.csd* [examples/STKBlowHole.csd]

*STKBowed.csd* [examples/STKBowed.csd]

*STKBrass.csd* [examples/STKBrass.csd]

*STKClarinet.csd* [examples/STKClarinet.csd]

*STKDrummer.csd* [examples/STKDrummer.csd]

*STKFMVoices.csd* [examples/STKFMVoices.csd]

*STKFlute.csd* [examples/STKFlute.csd]

*STKHevyMetl.csd* [examples/STKHevyMetl.csd]  
*STKMandolin.csd* [examples/STKMandolin.csd]  
*STKModalBar.csd* [examples/STKModalBar.csd]  
*STKMoog.csd* [examples/STKMoog.csd]  
*STKPercFlut.csd* [examples/STKPercFlut.csd]  
*STKPlucked.csd* [examples/STKPlucked.csd]  
*STKResonate.csd* [examples/STKResonate.csd]  
*STKRhodey.csd* [examples/STKRhodey.csd]  
*STKSaxofony.csd* [examples/STKSaxofony.csd]  
*STKShakers.csd* [examples/STKShakers.csd]  
*STKSimple.csd* [examples/STKSimple.csd]  
*STKSitar.csd* [examples/STKSitar.csd]  
*STKStifKarp.csd* [examples/STKStifKarp.csd]  
*STKTubeBell.csd* [examples/STKTubeBell.csd]  
*STKVoicForm.csd* [examples/STKVoicForm.csd]  
*STKWhistle.csd* [examples/STKWhistle.csd]  
*STKWurley.csd* [examples/STKWurley.csd]

**Signal Generators:Table Access.**

*oscil1i.csd* [examples/oscil1i.csd]  
*oscil1i.csd* [examples/oscil1i.csd]  
*ptable.csd* [examples/ptable.csd]  
*ptablei.csd* [examples/ptablei.csd]  
*tab.csd* [examples/tab.csd]  
*table.csd* [examples/table.csd]  
*tablei.csd* [examples/tablei.csd]

**Signal Generators:Wave Terrain Synthesis.**

*wterrain.csd* [examples/wterrain.csd]

**Signal Generators:Waveguide Physical Modeling.**

*pluck.csd* [examples/pluck.csd]  
*repluck.csd* [examples/repluck.csd]  
*streson.csd* [examples/streson.csd]  
*wgbow.csd* [examples/wgbow.csd]  
*wgbowedbar.csd* [examples/wgbowedbar.csd]  
*wgbrass.csd* [examples/wgbrass.csd]  
*wgclar.csd* [examples/wgclar.csd]  
*wgflute.csd* [examples/wgflute.csd]  
*wgpluck.csd* [examples/wgpluck.csd]  
*wgpluck\_brighter.csd* [examples/wgpluck\_brighter.csd]  
*wgpluck2.csd* [examples/wgpluck2.csd]

**Signal I/O:File I/O.**

*dumpk.csd* [examples/dumpk.csd]  
*dumpk-2.csd* [examples/dumpk-2.csd]  
*dumpk2.csd* [examples/dumpk2.csd]  
*dumpk3.csd* [examples/dumpk3.csd]  
*dumpk4.csd* [examples/dumpk4.csd]  
*ficlose.csd* [examples/ficlose.csd]  
*fin.csd* [examples/fin.csd]  
*fini.csd* [examples/fini.csd]  
*fink.csd* [examples/fink.csd]  
*fiopen.csd* [examples/fiopen.csd]  
*fout.csd* [examples/fout.csd]  
*fout\_ftable.csd* [examples/fout\_ftable.csd]  
*fouti.csd* [examples/fouti.csd]  
*foutir.csd* [examples/foutir.csd]  
*fprintks.csd* [examples/fprintks.csd]  
*fprintks-2.csd* [examples/fprintks-2.csd]  
*scogen.csd* [examples/scogen.csd]

*fprints.csd* [examples/fprints.csd]

*hdf5read.csd* [examples/hdf5read.csd]

*hdf5write.csd* [examples/hdf5write.csd]

*readf.csd* [examples/readf.csd]

*readfi.csd* [examples/readfi.csd]

*readk.csd* [examples/readk.csd]

*readk-2.csd* [examples/readk-2.csd]

*readk2.csd* [examples/readk2.csd]

*readk3.csd* [examples/readk3.csd]

*readk4.csd* [examples/readk4.csd]

**Signal I/O:Signal Input.**

*diskin.csd* [examples/diskin.csd]

*diskin2.csd* [examples/diskin2.csd]

*in.csd* [examples/in.csd]

*inch.csd* [examples/inch.csd]

*inq.csd* [examples/inq.csd]

*inrg.csd* [examples/inrg.csd]

*ins.csd* [examples/ins.csd]

*invalue.csd* [examples/invalue.csd]

*mp3in.csd* [examples/mp3in.csd]

*soundin.csd* [examples/soundin.csd]

**Signal I/O:Signal Output.**

*mdelay.csd* [examples/mdelay.csd]

*monitor.csd* [examples/monitor.csd]

*out.csd* [examples/out.csd]

*outc.csd* [examples/outc.csd]

*outch.csd* [examples/outch.csd]

*outch-2.csd* [examples/outch-2.csd]

*outq.csd* [examples/outq.csd]

*outq1.csd* [examples/outq1.csd]

*outq2.csd* [examples/outq2.csd]

*outq3.csd* [examples/outq3.csd]

*outq4.csd* [examples/outq4.csd]

*outrg.csd* [examples/outrg.csd]

*outs.csd* [examples/outs.csd]

*outs1.csd* [examples/outs1.csd]

*outs2.csd* [examples/outs2.csd]

*outvalue.csd* [examples/outvalue.csd]

**Signal I/O:Software Bus.**

*chnclear.csd* [examples/chnclear.csd]

*chnget.csd* [examples/chnget.csd]

*chnmix.csd* [examples/chnmix.csd]

*chnset.csd* [examples/chnset.csd]

**Signal I/O:Printing and Display.**

*dispfft.csd* [examples/dispfft.csd]

*display.csd* [examples/display.csd]

*flashtxt.csd* [examples/flashtxt.csd]

*print.csd* [examples/print.csd]

*printf.csd* [examples/printf.csd]

*printk.csd* [examples/printk.csd]

*printk2.csd* [examples/printk2.csd]

*printks.csd* [examples/printks.csd]

*printks2.csd* [examples/printks2.csd]

*prints.csd* [examples/prints.csd]

**Signal I/O:Soundfile Queries.**

*filebit.csd* [examples/filebit.csd]

*filelen.csd* [examples/filelen.csd]

*filenchnls.csd* [examples/filenchnls.csd]

*filepeak.csd* [examples/filepeak.csd]

*filesr.csd* [examples/filesr.csd]

*filevalid.csd* [examples/filevalid.csd]

*mp3len.csd* [examples/mp3len.csd]

**Signal Modifiers:Amplitude Modifiers.**

*balance.csd* [examples/balance.csd]

*clip.csd* [examples/clip.csd]

*compress.csd* [examples/compress.csd]

*compress2.csd* [examples/compress2.csd]

*dam.csd* [examples/dam.csd]

*dam\_expanded.csd* [examples/dam\_expanded.csd]

*gain.csd* [examples/gain.csd]

**Signal Modifiers:Convolution and Morphing.**

*convolve.csd* [examples/convolve.csd]

*cross2.csd* [examples/cross2.csd]

*dconv.csd* [examples/dconv.csd]

*ftconv.csd* [examples/ftconv.csd]

*ftmorf.csd* [examples/ftmorf.csd]

*liveconv.csd* [examples/liveconv.csd]

*pconvolve.csd* [examples/pconvolve.csd]

*tvconv.csd* [examples/tvconv.csd]

**Signal Modifiers:Delay.**

*delay.csd* [examples/delay.csd]

*delayl.csd* [examples/delayl.csd]

*delayk.csd* [examples/delayk.csd]

*delayr.csd* [examples/delayr.csd]

*delayw.csd* [examples/delayw.csd]

*deltap.csd* [examples/deltap.csd]

*deltap3.csd* [examples/deltap3.csd]

*deltapi.csd* [examples/deltapi.csd]

*deltapn.csd* [examples/deltapn.csd]

*deltapx.csd* [examples/deltapx.csd]

*deltapxw.csd* [examples/deltapxw.csd]

*multitap.csd* [examples/multitap.csd]

*vdelay.csd* [examples/vdelay.csd]

*vdelay3.csd* [examples/vdelay3.csd]

*vdelayx.csd* [examples/vdelayx.csd]

*vdelayxq.csd* [examples/vdelayxq.csd]

*vdelayxs.csd* [examples/vdelayxs.csd]

*vdelayxw.csd* [examples/vdelayxw.csd]

*vdelayxwq.csd* [examples/vdelayxwq.csd]

*vdelayxws.csd* [examples/vdelayxws.csd]

**Signal Modifiers:Panning and Spatialization.**

*bformenc.csd* [examples/bformenc.csd]

*bformenc1.csd* [examples/bformenc1.csd]

*bformenc.csd* [examples/bformenc.csd]

*bformenc1.csd* [examples/bformenc1.csd]

*hrtfearly.csd* [examples/hrtfearly.csd]

*hrtfer.csd* [examples/hrtfer.csd]

*hrtfmove.csd* [examples/hrtfmove.csd]

*hrtfmove2.csd* [examples/hrtfmove2.csd]

*hrtfearly.csd* [examples/hrtfearly.csd]

*hrtfstat.csd* [examples/hrtfstat.csd]

*hrtfstat-2.csd* [examples/hrtfstat-2.csd]

*locsend\_stereo.csd* [examples/locsend\_stereo.csd]

*locsig\_quad.csd* [examples/locsig\_quad.csd]

*pan.csd* [examples/pan.csd]  
*pan2.csd* [examples/pan2.csd]  
*space\_quad.csd* [examples/space\_quad.csd]  
*space\_stereo.csd* [examples/space\_stereo.csd]  
*spat3d\_stereo.csd* [examples/spat3d\_stereo.csd]  
*spat3d\_UHJ.csd* [examples/spat3d\_UHJ.csd]  
*spat3d\_quad.csd* [examples/spat3d\_quad.csd]  
*spat3dt.csd* [examples/spat3dt.csd]  
*spdist.csd* [examples/spdist.csd]  
*spsend.csd* [examples/spsend.csd]  
*vbap.csd* [examples/vbap.csd]  
*vbap4.csd* [examples/vbap4.csd]  
*vbap4move.csd* [examples/vbap4move.csd]  
*vbap8.csd* [examples/vbap8.csd]  
*vbap8move.csd* [examples/vbap8move.csd]  
*vbapg.csd* [examples/vbapg.csd]  
*vbapgmove.csd* [examples/vbapgmove.csd]  
*vbaplsinit.csd* [examples/vbaplsinit.csd]  
*vbapmove.csd* [examples/vbapmove.csd]

**Signal Modifiers:Reverberation.**

*alpass.csd* [examples/alpass.csd]  
*babo.csd* [examples/babo.csd]  
*babo\_expert.csd* [examples/babo\_expert.csd]  
*comb.csd* [examples/comb.csd]  
*combinv.csd* [examples/combinv.csd]  
*freeverb.csd* [examples/freeverb.csd]  
*nestedap.csd* [examples/nestedap.csd]  
*nreverb.csd* [examples/nreverb.csd]  
*nreverb\_ftable.csd* [examples/nreverb\_ftable.csd]

*plate.csd* [examples/plate.csd]

*reverb.csd* [examples/reverb.csd]

*reverbsc.csd* [examples/reverbsc.csd]

*valpass.csd* [examples/valpass.csd]

*valpass-2.csd* [examples/valpass-2.csd]

*vcomb.csd* [examples/vcomb.csd]

**Signal Modifiers:Sample Level Operators.**

*denorm.csd* [examples/denorm.csd]

*diff.csd* [examples/diff.csd]

*downsamp.csd* [examples/downsamp.csd]

*fold.csd* [examples/fold.csd]

*integ.csd* [examples/integ.csd]

*interp.csd* [examples/interp.csd]

*ntrpol.csd* [examples/ntrpol.csd]

*opa.csd* [examples/opa.csd]

*samphold.csd* [examples/samphold.csd]

*upsamp.csd* [examples/upsamp.csd]

*vaet.csd* [examples/vaet.csd]

*vaset.csd* [examples/vaset.csd]

**Signal Modifiers:Signal Limiters.**

*limit.csd* [examples/limit.csd]

*mirror.csd* [examples/mirror.csd]

*wrap.csd* [examples/wrap.csd]

**Signal Modifiers:Special Effects.**

*distort.csd* [examples/distort.csd]

*distort1.csd* [examples/distort1.csd]

*flanger.csd* [examples/flanger.csd]

*harmon.csd* [examples/harmon.csd]

*harmon3.csd* [examples/harmon3.csd]

*phaser1.csd* [examples/phaser1.csd]

*phaser2.csd* [examples/phaser2.csd]

**Signal Modifiers:Standard Filters.**

*atone.csd* [examples/atone.csd]

*atonex.csd* [examples/atonex.csd]

*biquad.csd* [examples/biquad.csd]

*biquad-2.csd* [examples/biquad-2.csd]

*biquada.csd* [examples/biquada.csd]

*butterbp.csd* [examples/butterbp.csd]

*butterbr.csd* [examples/butterbr.csd]

*butterhp.csd* [examples/butterhp.csd]

*butterlp.csd* [examples/butterlp.csd]

*clfilt\_lowpass.csd* [examples/clfilt\_lowpass.csd]

*clfilt\_highpass.csd* [examples/clfilt\_highpass.csd]

*diode\_ladder.csd* [examples/diode\_ladder.csd]

*doppler.csd* [examples/doppler.csd]

*k35.csd* [examples/k35.csd]

*k35.csd* [examples/k35.csd]

*median.csd* [examples/median.csd]

*mediank.csd* [examples/mediank.csd]

*mode.csd* [examples/mode.csd]

*tone.csd* [examples/tone.csd]

*tonex.csd* [examples/tonex.csd]

*zdf\_1pole.csd* [examples/zdf\_1pole.csd]

*zdf\_1pole\_mode.csd* [examples/zdf\_1pole\_mode.csd]

*zdf\_2pole.csd* [examples/zdf\_2pole.csd]

*zdf\_2pole\_mode.csd* [examples/zdf\_2pole\_mode.csd]

*zdf\_ladder.csd* [examples/zdf\_ladder.csd]

**Signal Modifiers:Standard Filters:Resonant.**

*areson.csd* [examples/areson.csd]

*bqrez.csd* [examples/bqrez.csd]

*lowpass2.csd* [examples/lowpass2.csd]

*lowres.csd* [examples/lowres.csd]

*lowresx.csd* [examples/lowresx.csd]

*lpf18.csd* [examples/lpf18.csd]

*moogladder.csd* [examples/moogladder.csd]

*moogladder2.csd* [examples/moogladder2.csd]

*moogvcf.csd* [examples/moogvcf.csd]

*moogvcf2.csd* [examples/moogvcf2.csd]

*mvchpf.csd* [examples/mvchpf.csd]

*mvclpf1.csd* [examples/mvclpf1.csd]

*mvclpf2.csd* [examples/mvclpf2.csd]

*mvclpf3.csd* [examples/mvclpf3.csd]

*mvclpf4.csd* [examples/mvclpf4.csd]

*reson.csd* [examples/reson.csd]

*resonr.csd* [examples/resonr.csd]

*resonx.csd* [examples/resonx.csd]

*resony.csd* [examples/resony.csd]

*resonr.csd* [examples/resonr.csd]

*rezzy.csd* [examples/rezzy.csd]

*statevar.csd* [examples/statevar.csd]

*svfilter.csd* [examples/svfilter.csd]

*tbvcf.csd* [examples/tbvcf.csd]

*vlowres.csd* [examples/vlowres.csd]

**Signal Modifiers:Standard Filters:Control.**

*aresonk.csd* [examples/aresonk.csd]

*atonek.csd* [examples/atonek.csd]

*lineto.csd* [examples/lineto.csd]

*port.csd* [examples/port.csd]

*portk.csd* [examples/portk.csd]

*resonk.csd* [examples/resonk.csd]

*resonxk.csd* [examples/resonxk.csd]

*sc\_lag.csd* [examples/sc\_lag.csd]

*sc\_lagud.csd* [examples/sc\_lagud.csd]

*sc\_trig.csd* [examples/sc\_trig.csd]

*tlineto.csd* [examples/tlineto.csd]

*tonek.csd* [examples/tonek.csd]

**Signal Modifiers:Specialized Filters.**

*dcblock.csd* [examples/dcblock.csd]

*dcblock2.csd* [examples/dcblock2.csd]

*eqfil.csd* [examples/eqfil.csd]

*filter2.csd* [examples/filter2.csd]

*fmanal.csd* [examples/fmanal.csd]

*fofilter.csd* [examples/fofilter.csd]

*hilbert.csd* [examples/hilbert.csd]

*hilbert\_barberpole.csd* [examples/hilbert\_barberpole.csd]

*hilbert2.csd* [examples/hilbert2.csd]

*nlfilt.csd* [examples/nlfilt.csd]

*nlfilt2.csd* [examples/nlfilt2.csd]

*pareq.csd* [examples/pareq.csd]

*rbjeq.csd* [examples/rbjeq.csd]

**Signal Modifiers:Waveguides.**

*wguide1.csd* [examples/wguide1.csd]

*wguide2.csd* [examples/wguide2.csd]

**Signal Modifiers:Waveshaping.**

*chebyshevpoly.csd* [examples/chebyshevpoly.csd]

*pdclip.csd* [examples/pdclip.csd]

*pdhalf.csd* [examples/pdhalf.csd]

*pdhalfy.csd* [examples/pdhalfy.csd]

*powershape.csd* [examples/powershape.csd]

**Signal Modifiers:Comparators and Accumulators.**

*cmp.csd* [examples/cmp.csd]

*max.csd* [examples/max.csd]

*max\_k.csd* [examples/max\_k.csd]

*maxabs.csd* [examples/maxabs.csd]

*maxabsaccum.csd* [examples/maxabsaccum.csd]

*maxaccum.csd* [examples/maxaccum.csd]

*min.csd* [examples/min.csd]

*minabs.csd* [examples/minabs.csd]

*minabsaccum.csd* [examples/minabsaccum.csd]

*minaccum.csd* [examples/minaccum.csd]

**Instrument Control:Clock Control.**

*clockoff.csd* [examples/clockoff.csd]

*clockon.csd* [examples/clockon.csd]

**Instrument Control:Conditional Values.**

*equals.csd* [examples>equals.csd]

*greaterequal.csd* [examples/greaterequal.csd]

*greaterthan.csd* [examples/greaterthan.csd]

*lessequal.csd* [examples/lessequal.csd]

*lessthan.csd* [examples/lessthan.csd]

*notequal.csd* [examples/notequal.csd]

**Instrument Control:Compilation.**

*compilecsd.csd* [examples/compilecsd.csd]

*compileorc.csd* [examples/compileorc.csd]

*compilestr.csd* [examples/compilestr.csd]

**Instrument Control:Duration Control.**

*ihold.csd* [examples/ihold.csd]

*turnoff.csd* [examples/turnoff.csd]

*turnoff2.csd* [examples/turnoff2.csd]

**Instrument Control:Invocation.**

*event.csd* [examples/event.csd]

*event\_named.csd* [examples/event\_named.csd]

*event\_i.csd* [examples/event\_i.csd]

*mute.csd* [examples/mute.csd]

*nstance.csd* [examples/nstance.csd]

*readscore.csd* [examples/readscore.csd]

*shedkwhen.csd* [examples/schedkwhen.csd]

*shedkwhennamed.csd* [examples/schedkwhennamed.csd]

*schedule.csd* [examples/schedule.csd]

*shedwhen.csd* [examples/schedwhen.csd]

*scoreline.csd* [examples/scoreline.csd]

*scoreline\_i.csd* [examples/scoreline\_i.csd]

**Instrument Control:Program Flow Control.**

*cggoto.csd* [examples/cggoto.csd]

*cigoto.csd* [examples/cigoto.csd]

*ckgoto.csd* [examples/ckgoto.csd]

*cngoto.csd* [examples/cngoto.csd]

*else.csd* [examples/else.csd]

*elseif.csd* [examples/elseif.csd]

*endif.csd* [examples/endif.csd]

*goto.csd* [examples/goto.csd]

*igoto.csd* [examples/igoto.csd]

*kgoto.csd* [examples/kgoto.csd]

*ifthen.csd* [examples/ifthen.csd]

*igoto.csd* [examples/igoto.csd]

*kgoto.csd* [examples/kgoto.csd]

*loop\_le.csd* [examples/loop\_le.csd]

*loop\_lt.csd* [examples/loop\_lt.csd]

*tigoto.csd* [examples/tigoto.csd]

*timeout.csd* [examples/timeout.csd]

*until.csd* [examples/until.csd]

*while.csd* [examples/while.csd]

#### **Instrument Control:Realtime Performance Control.**

*active.csd* [examples/active.csd]

*active\_k.csd* [examples/active\_k.csd]

*active\_scale.csd* [examples/active\_scale.csd]

*cpumeter.csd* [examples/cpumeter.csd]

*cpuprc.csd* [examples/cpuprc.csd]

*exitnow.csd* [examples/exitnow.csd]

*jacktransport.csd* [examples/jacktransport.csd]

*maxalloc.csd* [examples/maxalloc.csd]

*prealloc.csd* [examples/prealloc.csd]

#### **Instrument Control:Initialization and Reinitialization.**

*assign.csd* [examples/assign.csd]

*init.csd* [examples/init.csd]

*p.csd* [examples/p.csd]

*passign.csd* [examples/passign.csd]

*reverb.csd* [examples/reverb.csd]

*pset.csd* [examples/pset.csd]

*pset-midi.csd* [examples/pset-midi.csd]

*reinit.csd* [examples/reinit.csd]

*reinit.csd* [examples/reinit.csd]

*tival.csd* [examples/tival.csd]

**Instrument Control:Sensing and Control.**

*checkbox.csd* [examples/checkbox.csd]

*changed.csd* [examples/changed.csd]

*changed2.csd* [examples/changed2.csd]

*changed2a.csd* [examples/changed2a.csd]

*checkbox.csd* [examples/checkbox.csd]

*follow.csd* [examples/follow.csd]

*follow2.csd* [examples/follow2.csd]

*getcfg.csd* [examples/getcfg.csd]

*joystick.csd* [examples/joystick.csd]

*joystick-2.csd* [examples/joystick-2.csd]

*metro.csd* [examples/metro.csd]

*metro-2.csd* [examples/metro-2.csd]

*miditempo.csd* [examples/miditempo.csd]

*p5g.csd* [examples/p5g.csd]

*pcount.csd* [examples/pcount.csd]

*peak.csd* [examples/peak.csd]

*pindex.csd* [examples/pindex.csd]

*pindex-2.csd* [examples/pindex-2.csd]

*pitch.csd* [examples/pitch.csd]

*pitchamdf.csd* [examples/pitchamdf.csd]

*plltrack.csd* [examples/plltrack.csd]

*ptrack.csd* [examples/ptrack.csd]

*readscratch.csd* [examples/readscratch.csd]

*rewindscore.csd* [examples/rewindscore.csd]

*rms.csd* [examples/rms.csd]

*sensekey.csd* [examples/sensekey.csd]

*FLpanel-sensekey.csd* [examples/FLpanel-sensekey.csd]

*FLpanel-sensekey2.csd* [examples/FLpanel-sensekey2.csd]

*seqtime.csd* [examples/seqtime.csd]

*seqtime2.csd* [examples/seqtime2.csd]

*setctrl.csd* [examples/setctrl.csd]

*setscorepos.csd* [examples/setscorepos.csd]

*tempest.csd* [examples/tempest.csd]

*tempo.csd* [examples/tempo.csd]

*tempoval.csd* [examples/tempoval.csd]

*timedseq.csd* [examples/timedseq.csd]

*trigger.csd* [examples/trigger.csd]

*trigseq.csd* [examples/trigseq.csd]

*vactrol.csd* [examples/vactrol.csd]

*wii.csd* [examples/wii.csd]

*readscratch.csd* [examples/readscratch.csd]

*xyin.csd* [examples/xyin.csd]

#### **Instrument Control:Stacks.**

*pop.csd* [examples/pop.csd]

*push.csd* [examples/push.csd]

*stack.csd* [examples/stack.csd]

#### **Instrument Control:Subinstrument Control.**

*subinstr.csd* [examples/subinstr.csd]

*subinstr\_named.csd* [examples/subinstr\_named.csd]

#### **Instrument Control:Time Reading.**

*date.csd* [examples/date.csd]

*dates.csd* [examples/dates.csd]

*readclock.csd* [examples/readclock.csd]

*rtclock.csd* [examples/rtclock.csd]

*timeinstk.csd* [examples/timeinstk.csd]

*timeinsts.csd* [examples/timeinsts.csd]

*timek.csd* [examples/timek.csd]

*times\_complex.csd* [examples/times\_complex.csd]

### **Jacko Opcodes.**

*JackoInfo.csd* [examples/JackoInfo.csd]

*JackoInit.csd* [examples/JackoInit.csd]

### **Lua Opcodes.**

*luaopcode.csd* [examples/luaopcode.csd]

*luamoog.csd* [examples/luamoog.csd]

### **Serial I/O.**

*serialBegin.csd* [examples/serialBegin.csd]

*serialRead.csd* [examples/serialRead.csd]

*serialWrite.csd* [examples/serialWrite.csd]

### **Table Control.**

*ftfree.csd* [examples/ftfree.csd]

*ftgen.csd* [examples/ftgen.csd]

*ftgen-2.csd* [examples/ftgen-2.csd]

*ftgentmp.csd* [examples/ftgentmp.csd]

*getftargs.csd* [examples/getftargs.csd]

### **Table Control:Table Queries.**

*array.csd* [examples/array.csd]

*fillarray.csd* [examples/fillarray.csd]

*ftchnls.csd* [examples/ftchnls.csd]

*ftcps.csd* [examples/ftcps.csd]

*ftlen.csd* [examples/ftlen.csd]

*ftlptim.csd* [examples/ftlptim.csd]

*ftsr.csd* [examples/ftsr.csd]

*genarray.csd* [examples/genarray.csd]

*genarray\_i.csd* [examples/genarray\_i.csd]

*lenarray.csd* [examples/lenarray.csd]

*maparray.csd* [examples/maparray.csd]

*nsamp.csd* [examples/nsamp.csd]

*slicearray.csd* [examples/slicearray.csd]

*tableng.csd* [examples/tableng.csd]

*tabsum.csd* [examples/tabsum.csd]

**Table Control:Dynamic Selection.**

*tableikt.csd* [examples/tableikt.csd]

*tablekt.csd* [examples/tablekt.csd]

*tablexkt.csd* [examples/tablexkt.csd]

**Table Control:Read/Write Opreations.**

*ftsamplbank.csd* [examples/ftsamplbank.csd]

*ftsav.csd* [examples/ftsav.csd]

*tablecopy.csd* [examples/tablecopy.csd]

*tablefilter.csd* [examples/tablefilter.csd]

*tablefilter.csd* [examples/tablefilter.csd]

*tableimix.csd* [examples/tableimix.csd]

*tableiw.csd* [examples/tableiw.csd]

*tablemix.csd* [examples/tablemix.csd]

*tabmorph.csd* [examples/tabmorph.csd]

*tabmorpha.csd* [examples/tabmorpha.csd]

*tabmorphak.csd* [examples/tabmorphak.csd]

*tabmorphi.csd* [examples/tabmorphi.csd]

**FLTK:Containers.**

*FLpanel.csd* [examples/FLpanel.csd]

*FLscroll.csd* [examples/FLscroll.csd]

*FLtabs.csd* [examples/FLtabs.csd]

**FLTK:Valuators.**

*FLcount.csd* [examples/FLcount.csd]

*FLjoy.csd* [examples/FLjoy.csd]

*FLknob.csd* [examples/FLknob.csd]

*FLknob-2.csd* [examples/FLknob-2.csd]

*FLroller.csd* [examples/FLroller.csd]

*FLslider.csd* [examples/FLslider.csd]

*FLslider-2.csd* [examples/FLslider-2.csd]

*FLtext.csd* [examples/FLtext.csd]

**FLTK:Other.**

*FLbox.csd* [examples/FLbox.csd]

*FLbutBank.csd* [examples/FLbutBank.csd]

*FLbutton.csd* [examples/FLbutton.csd]

*FLexecButton.csd* [examples/FLexecButton.csd]

*FLhvsBox.csd* [examples/FLhvsBox.csd]

*FLhvsBoxSetValue.csd* [examples/FLhvsBoxSetValue.csd]

*FLkeyIn.csd* [examples/FLkeyIn.csd]

*FLmouse.csd* [examples/FLmouse.csd]

*FLsavesnap\_simple.csd* [examples/FLsavesnap\_simple.csd]

*FLsavesnap.csd* [examples/FLsavesnap.csd]

*FLslidBnk.csd* [examples/FLslidBnk.csd]

*FLslidBnk2.csd* [examples/FLslidBnk2.csd]

*FLslidBnk2Set.csd* [examples/FLslidBnk2Set.csd]

*FLslidBnk2Setk.csd* [examples/FLslidBnk2Setk.csd]

*FLslidBnkGetHandle.csd* [examples/FLslidBnkGetHandle.csd]

*FLslidBnkSet.csd* [examples/FLslidBnkSet.csd]

*FLslidBnkSetk.csd* [examples/FLslidBnkSetk.csd]

*FLvalue.csd* [examples/FLvalue.csd]

*FLvslidBnk.csd* [examples/FLvslidBnk.csd]

*FLvslidBnk2.csd* [examples/FLvslidBnk2.csd]

*FLxyin.csd* [examples/FLxyin.csd]

*FLxyin-2.csd* [examples/FLxyin-2.csd]

*vphaseseg.csd* [examples/vphaseseg.csd]

**FLTK:Appearance.**

*FLsetcolor.csd* [examples/FLsetcolor.csd]

*FLsetText.csd* [examples/FLsetText.csd]

**Mathematical Operations:Arithmetic and Logic Operations.**

*adds.csd* [examples/adds.csd]

*divides.csd* [examples/divides.csd]

*modulus.csd* [examples/modulus.csd]

*multiplies.csd* [examples/multiplies.csd]

*opand.csd* [examples/opand.csd]

*bitwise.csd* [examples/bitwise.csd]

*bitshift.csd* [examples/bitshift.csd]

*logicOR.csd* [examples/logicOR.csd]

*raises.csd* [examples/raises.csd]

*subtracts.csd* [examples/subtracts.csd]

**Mathematical Operations:Comparators and Accumulators.**

*clear.csd* [examples/clear.csd]

*vincr.csd* [examples/vincr.csd]

*vincr-complex.csd* [examples/vincr-complex.csd]

**Mathematical Operations:Mathematical Functions.**

*abs.csd* [examples/abs.csd]

*ceil.csd* [examples/ceil.csd]

*exp.csd* [examples/exp.csd]

*floor.csd* [examples/floor.csd]

*frac.csd* [examples/frac.csd]

*int.csd* [examples/int.csd]

*log10.csd* [examples/log10.csd]

*log2.csd* [examples/log2.csd]

*logbtwo.csd* [examples/logbtwo.csd]

*powoftwo.csd* [examples/powoftwo.csd]

*qinf.csd* [examples/qinf.csd]

*qnan.csd* [examples/qnan.csd]

*round.csd* [examples/round.csd]

*sqrt.csd* [examples/sqrt.csd]

**Mathematical Operations: Trigonometric Functions.**

*cos.csd* [examples/cos.csd]

*cosh.csd* [examples/cosh.csd]

*cosinv.csd* [examples/cosinv.csd]

*signum.csd* [examples/signum.csd]

*sin.csd* [examples/sin.csd]

*sinh.csd* [examples/sinh.csd]

*sininv.csd* [examples/sininv.csd]

*tan.csd* [examples/tan.csd]

*tanh.csd* [examples/tanh.csd]

*taninv.csd* [examples/taninv.csd]

**Mathematical Operations: Amplitude Functions.**

*ampdb.csd* [examples/ampdb.csd]

*ampdbfs.csd* [examples/ampdbfs.csd]

*db.csd* [examples/db.csd]

*dbamp.csd* [examples/dbamp.csd]

*dbfsamp.csd* [examples/dbfsamp.csd]

**Mathematical Operations:Random Functions.**

*birnd.csd* [examples/birnd.csd]

*rnd.csd* [examples/rnd.csd]

**Mathematical Operations:Opcode Equivalent of Functions.**

*divz.csd* [examples/divz.csd]

*mac.csd* [examples/mac.csd]

*maca.csd* [examples/maca.csd]

*polynomial.csd* [examples/polynomial.csd]

*pow.csd* [examples/pow.csd]

*product.csd* [examples/product.csd]

*sum.csd* [examples/sum.csd]

*taninv2.csd* [examples/taninv2.csd]

**Pitch Converters:Functions.**

*cent.csd* [examples/cent.csd]

*cpsmidinn.csd* [examples/cpsmidinn.csd]

*cpsmidinn2.csd* [examples/cpsmidinn2.csd]

*cpsoct.csd* [examples/cpsoct.csd]

*cpspch.csd* [examples/cpspch.csd]

*mtof-ftom.csd* [examples/mtof-ftom.csd]

*mtof-ftom.csd* [examples/mtof-ftom.csd]

*mton-ntom.csd* [examples/mton-ntom.csd]

*mton-ntom.csd* [examples/mton-ntom.csd]

*octave.csd* [examples/octave.csd]

*octcps.csd* [examples/octcps.csd]

*cpsmidinn.csd* [examples/cpsmidinn.csd]

*octpch.csd* [examples/octpch.csd]

*cpsmidinn.csd* [examples/cpsmidinn.csd]

*pchoct.csd* [examples/pchoct.csd]

*pchtom.csd* [examples/pchtom.csd]

*semitone.csd* [examples/semitone.csd]

**Pitch Converters:Tuning Opcodes.**

*cps2pch.csd* [examples/cps2pch.csd]

*cps2pch\_ftable.csd* [examples/cps2pch\_ftable.csd]

*cps2pch\_19et.csd* [examples/cps2pch\_19et.csd]

*cpstun.csd* [examples/cpstun.csd]

*cpstuni.csd* [examples/cpstuni.csd]

*cpsxpch.csd* [examples/cpsxpch.csd]

*cpsxpch\_105et.csd* [examples/cpsxpch\_105et.csd]

*cpsxpch\_pierce.csd* [examples/cpsxpch\_pierce.csd]

**Real-time MIDI:Input.**

*aftouch.csd* [examples/aftouch.csd]

*chanctrl.csd* [examples/chanctrl.csd]

*ctrl14.csd* [examples/ctrl14.csd]

*ctrl21.csd* [examples/ctrl21.csd]

*ctrl7.csd* [examples/ctrl7.csd]

*initc7.csd* [examples/initc7.csd]

*massign.csd* [examples/massign.csd]

*midic7.csd* [examples/midic7.csd]

*midictrl.csd* [examples/midictrl.csd]

*notnum.csd* [examples/notnum.csd]

*pchbend.csd* [examples/pchbend.csd]

*pgmassign.csd* [examples/pgmassign.csd]

*pgmassign\_ignore.csd* [examples/pgmassign\_ignore.csd]

*pgmassign\_advanced.csd* [examples/pgmassign\_advanced.csd]

*polyaft.csd* [examples/polyaft.csd]

*veloc.csd* [examples/veloc.csd]

**Real-time MIDI:Output.**

*nrpn.csd* [examples/nrpn.csd]  
*outiat.csd* [examples/outiat.csd]  
*outic.csd* [examples/outic.csd]  
*outipb.csd* [examples/outipb.csd]  
*outipc.csd* [examples/outipc.csd]  
*outkat.csd* [examples/outkat.csd]  
*outkc.csd* [examples/outkc.csd]  
*outkpb.csd* [examples/outkpb.csd]  
*outkpc.csd* [examples/outkpc.csd]  
*outkpc\_fltk.csd* [examples/outkpc\_fltk.csd]

**Real-time MIDI:Converters.**

*ampmidi.csd* [examples/ampmidi.csd]  
*ampmidid.csd* [examples/ampmidid.csd]  
*cpsmidi.csd* [examples/cpsmidi.csd]  
*cpsmidib.csd* [examples/cpsmidib.csd]  
*cpstmid.csd* [examples/cpstmid.csd]  
*octmidi.csd* [examples/octmidi.csd]  
*octmidib.csd* [examples/octmidib.csd]  
*pchmidi.csd* [examples/pchmidi.csd]  
*pchmidib.csd* [examples/pchmidib.csd]

**Real-time MIDI:Generic I/O.**

*midiin.csd* [examples/midiin.csd]  
*midiout.csd* [examples/midiout.csd]

**Real-time MIDI:Event Extenders.**

*xtratim.csd* [examples/xtratim.csd]  
*xtratim-2.csd* [examples/xtratim-2.csd]

**Real-time MIDI:Note Output.**

*midion\_simple.csd* [examples/midion\_simple.csd]

*midion\_scale.csd* [examples/midion\_scale.csd]

*midion2.csd* [examples/midion2.csd]

*moscil.csd* [examples/moscil.csd]

*noteondur.csd* [examples/noteondur.csd]

*noteondur2.csd* [examples/noteondur2.csd]

### **Real-time MIDI:MIDI/Score Interoperability.**

*midichannelaftertouch.csd* [examples/midichannelaftertouch.csd]

*midichn.csd* [examples/midichn.csd]

*midichn\_advanced.csd* [examples/midichn\_advanced.csd]

*midicontrolchange.csd* [examples/midicontrolchange.csd]

*mididefault.csd* [examples/mididefault.csd]

*midinoteoff.csd* [examples/midinoteoff.csd]

*midinoteoncps.csd* [examples/midinoteoncps.csd]

*midinoteonkey.csd* [examples/midinoteonkey.csd]

*midinoteonoct.csd* [examples/midinoteonoct.csd]

*midinoteonpch.csd* [examples/midinoteonpch.csd]

*midipitchbend.csd* [examples/midipitchbend.csd]

*midipolyaftertouch.csd* [examples/midipolyaftertouch.csd]

*midiprogramchange.csd* [examples/midiprogramchange.csd]

### **Real-time MIDI:System Realtime.**

*mclock.csd* [examples/mclock.csd]

### **Signal Flow Graph Opcodes.**

*alwayson.csd* [examples/alwayson.csd]

*connect.csd* [examples/connect.csd]

*ftgenonce.csd* [examples/ftgenonce.csd]

*inleta.csd* [examples/inleta.csd]

*inletk.csd* [examples/inletk.csd]

*outleta.csd* [examples/outleta.csd]

*outletk.csd* [examples/outletk.csd]

**Spectral Processing:STFT.**

*pvadd.csd* [examples/pvadd.csd]

*pvbufread.csd* [examples/pvbufread.csd]

*pvcross.csd* [examples/pvcross.csd]

*pvinterp.csd* [examples/pvinterp.csd]

*pvoc.csd* [examples/pvoc.csd]

*pvread.csd* [examples/pvread.csd]

*tableseg.csd* [examples/tableseg.csd]

*tablexseg.csd* [examples/tablexseg.csd]

*vpvoc.csd* [examples/vpvoc.csd]

**Spectral Processing:LPC.**

*lpfreson.csd* [examples/lpfreson.csd]

*lpread.csd* [examples/lpread.csd]

*lpreson.csd* [examples/lpreson.csd]

*lpreson-2.csd* [examples/lpreson-2.csd]

**Spectral Processing:Streaming.**

*binit.csd* [examples/binit.csd]

*culdanal.csd* [examples/cudanal.csd]

*culdasliding.csd* [examples/cudasliding.csd]

*culdasynth.csd* [examples/cudasynth.csd]

*partials.csd* [examples/partials.csd]

*pvsadsyn.csd* [examples/pvsadsyn.csd]

*pvsanal.csd* [examples/pvsanal.csd]

*pvsarp.csd* [examples/pvsarp.csd]

*pvsarp2.csd* [examples/pvsarp2.csd]

*pvsbandp.csd* [examples/pvsbandp.csd]

*pvsbandr.csd* [examples/pvsbandr.csd]

*pvsbin.csd* [examples/pvsbin.csd]  
*pvsblur.csd* [examples/pvsblur.csd]  
*pvsbufread.csd* [examples/pvsbufread.csd]  
*pvsbufread2.csd* [examples/pvsbufread2.csd]  
*pvscale.csd* [examples/pvscale.csd]  
*pvscent.csd* [examples/pvscent.csd]  
*pvsceps.csd* [examples/pvsceps.csd]  
*pvcross.csd* [examples/pvcross.csd]  
*pvsdiskin.csd* [examples/pvsdiskin.csd]  
*pvsdisp.csd* [examples/pvsdisp.csd]  
*pvsfilter.csd* [examples/pvsfilter.csd]  
*pvsfread.csd* [examples/pvsfread.csd]  
*pvsfreeze.csd* [examples/pvsfreeze.csd]  
*pvsftr.csd* [examples/pvsftr.csd]  
*pvsftw.csd* [examples/pvsftw.csd]  
*pvsfwrite.csd* [examples/pvsfwrite.csd]  
*pvsgain.csd* [examples/pvsgain.csd]  
*pvsshift.csd* [examples/pvsshift.csd]  
*pvsifd.csd* [examples/pvsifd.csd]  
*pvsinfo.csd* [examples/pvsinfo.csd]  
*pvslock.csd* [examples/pvslock.csd]  
*pvsmaska.csd* [examples/pvsmaska.csd]  
*pvsmix.csd* [examples/pvsmix.csd]  
*pvsMOOTH.csd* [examples/pvsMOOTH.csd]  
*pvsMORPH.csd* [examples/pvsMORPH.csd]  
*pvsMORPH2.csd* [examples/pvsMORPH2.csd]  
*pvsosc.csd* [examples/pvsosc.csd]  
*pvspitch.csd* [examples/pvspitch.csd]  
*pvsstanal.csd* [examples/pvsstanal.csd]  
*pvsTRACE.csd* [examples/pvsTRACE.csd]

*pvsvoc.csd* [examples/pvsvoc.csd]

*pvswarp.csd* [examples/pvswarp.csd]

*pvsynth.csd* [examples/pvsynth.csd]

*resyn.csd* [examples/resyn.csd]

*sinsyn.csd* [examples/sinsyn.csd]

*tabifd.csd* [examples/tabifd.csd]

*tradsyn.csd* [examples/tradsyn.csd]

*trcross.csd* [examples/trcross.csd]

*trfilter.csd* [examples/trfilter.csd]

*trhighest.csd* [examples/trhighest.csd]

*trlowest.csd* [examples/trlowest.csd]

*trmix.csd* [examples/trmix.csd]

*trscale.csd* [examples/trscale.csd]

*trshift.csd* [examples/trshift.csd]

*trsplitted.csd* [examples/trsplitted.csd]

### **Spectral Processing:ATS.**

*ATSadd.csd* [examples/ATSadd.csd]

*ATSadd-2.csd* [examples/ATSadd-2.csd]

*ATSaddnz.csd* [examples/ATSaddnz.csd]

*ATSaddnz-2.csd* [examples/ATSaddnz-2.csd]

*ATSbufread.csd* [examples/ATSbufread.csd]

*ATScross.csd* [examples/ATScross.csd]

*ATSinfo.csd* [examples/ATSinfo.csd]

*ATSinfo-2.csd* [examples/ATSinfo-2.csd]

*ATSinterpread.csd* [examples/ATSinterpread.csd]

*ATSpartialtap.csd* [examples/ATSpartialtap.csd]

*ATSread.csd* [examples/ATSread.csd]

*ATSread-2.csd* [examples/ATSread-2.csd]

*ATSreadnz.csd* [examples/ATSreadnz.csd]

*ATSsinnoi.csd* [examples/ATSsinnoi.csd]

*ATSsinnoi-2.csd* [examples/ATSsinnoi-2.csd]

**Spectral Processing:Loris.**

*lorismorph.csd* [examples/lorismorph.csd]

*lorisplay.csd* [examples/lorisplay.csd]

*lorisread.csd* [examples/lorisread.csd]

**Spectral Processing:Other.**

*centroid.csd* [examples/centroid.csd]

*filesca.csd* [examples/filesca.csd]

*mincer.csd* [examples/mincer.csd]

*mp3sca.csd* [examples/mp3sca.csd]

*paulstretch.csd* [examples/paulstretch.csd]

*temposca.csd* [examples/temposca.csd]

**Strings:Definition.**

*strfromurl.csd* [examples/strfromurl.csd]

*strget.csd* [examples/strget.csd]

*strset.csd* [examples/strset.csd]

**Strings:Manipulation.**

*puts.csd* [examples/puts.csd]

*sprintf.csd* [examples/sprintf.csd]

*sprintfk.csd* [examples/sprintfk.csd]

*strcat.csd* [examples/strcat.csd]

*strcatk.csd* [examples/strcatk.csd]

*strcmp.csd* [examples/strcmp.csd]

*strcpyk.csd* [examples/strcpyk.csd]

*strindexk.csd* [examples/strindexk.csd]

*strrindex.csd* [examples/strrindex.csd]

*strsub.csd* [examples/strsub.csd]

**Strings:Conversion.**

*strchar.csd* [examples/strchar.csd]

*strlower.csd* [examples/strlower.csd]

*strtod.csd* [examples/strtod.csd]

*strtodk.csd* [examples/strtodk.csd]

*strtol.csd* [examples/strtol.csd]

*strtolk.csd* [examples/strtolk.csd]

**Vectorial:Tables.**

*vtable1k.csd* [examples/vtable1k.csd]

*vtablei.csd* [examples/vtablei.csd]

*vtablek.csd* [examples/vtablek.csd]

*vtablewa.csd* [examples/vtablewa.csd]

*vtablewk.csd* [examples/vtablewk.csd]

**Vectorial:Scalar operations.**

*vadd.csd* [examples/vadd.csd]

*vadd\_i.csd* [examples/vadd\_i.csd]

*vexp.csd* [examples/vexp.csd]

*vexp\_i.csd* [examples/vexp\_i.csd]

*vmult-2.csd* [examples/vmult-2.csd]

*vmult.csd* [examples/vmult.csd]

*vmult\_i.csd* [examples/vmult\_i.csd]

*vpow.csd* [examples/vpow.csd]

*vpow-2.csd* [examples/vpow-2.csd]

*vpow\_i.csd* [examples/vpow\_i.csd]

*vpow\_i-2.csd* [examples/vpow\_i-2.csd]

**Vectorial:Vectorial operations.**

*vaddv.csd* [examples/vaddv.csd]

*vcopy.csd* [examples/vcopy.csd]

*vdivv.csd* [examples/vdivv.csd]

*vexpv.csd* [examples/vexpv.csd]

*vmap.csd* [examples/vmap.csd]

*vmultv.csd* [examples/vmultv.csd]

*vpowv.csd* [examples/vpowv.csd]

*vsubv.csd* [examples/vsubv.csd]

**Vectorial:Envelopes.**

*vexpseg.csd* [examples/vexpseg.csd]

*vlinseg.csd* [examples/vlinseg.csd]

**Vectorial:Random.**

*vrandh.csd* [examples/vrandh.csd]

*vrandi.csd* [examples/vrandi.csd]

**Vectorial:Cellular Automata.**

*cell.csd* [examples/cell.csd]

*vcella.csd* [examples/vcella.csd]

**Zak Patch System.**

*zacl.csd* [examples/zacl.csd]

*zakinit.csd* [examples/zakinit.csd]

*zamod.csd* [examples/zamod.csd]

*zar.csd* [examples/zar.csd]

*zarg.csd* [examples/zarg.csd]

*zaw.csd* [examples/zaw.csd]

*zawm.csd* [examples/zawm.csd]

*zir.csd* [examples/zir.csd]

*ziw.csd* [examples/ziw.csd]

*ziwm.csd* [examples/ziwm.csd]

*zkcl.csd* [examples/zkcl.csd]

*zkmod.csd* [examples/zkmod.csd]

*zkr.csd* [examples/zkr.csd]

*zkw.csd* [examples/zkw.csd]

*zkwm.csd* [examples/zkwm.csd]

**Plugin Hosting:DSSI and LADSPA.**

*dssiactivate.csd* [examples/dssiactivate.csd]

*dssiaudio.csd* [examples/dssiaudio.csd]

*dssictls.csd* [examples/dssictls.csd]

*dssiinit.csd* [examples/dssiinit.csd]

*dssilist.csd* [examples/dssilist.csd]

**Plugin Hosting:VST.**

*vst4cs.csd* [examples/vst4cs.csd]

*vst4cs.csd* [examples/vst4cs.csd]

*vst4cs.csd* [examples/vst4cs.csd]

*vst4cs.csd* [examples/vst4cs.csd]

*vst4cs.csd* [examples/vst4cs.csd]

*vst4cs.csd* [examples/vst4cs.csd]

**OSC.**

*OSCmidisend.csd* [examples/OSCmidisend.csd]

*OSCmidircv.csd* [examples/OSCmidircv.csd]

*OSCrw.csd* [examples/OSCrw.csd]

**Faust Opcodes.**

*faustaudio.csd* [examples/faustaudio.csd]

*faustcompile.csd* [examples/faustcompile.csd]

*faustctl.csd* [examples/faustctl.csd]

*faustgen.csd* [examples/faustgen.csd]

**Remote Opcodes.**

*insremot.csd* [examples/insremot.csd]

*insremotM.csd* [examples/insremotM.csd]

*midremot.csd* [examples/midremot.csd]

**Mixer Opcodes.**

*Mixer.csd* [examples/Mixer.csd]

*Mixer.csd* [examples/Mixer.csd]

*Mixer.csd* [examples/Mixer.csd]

*Mixer.csd* [examples/Mixer.csd]

*Mixer.csd* [examples/Mixer.csd]

**Ableton Link Opcodes.**

*ableton\_link.csd* [examples/ableton\_link.csd]

**Image Processing Opcodes.**

*imageopcodes.csd* [examples/imageopcodes.csd]

*imageopcodes.csd* [examples/imageopcodes.csd]

*imageopcodesdemo2.csd* [examples/imageopcodesdemo2.csd]

*imageopcodes.csd* [examples/imageopcodes.csd]

*imageopcodes.csd* [examples/imageopcodes.csd]

*imageopcodes.csd* [examples/imageopcodes.csd]

*imageopcodes.csd* [examples/imageopcodes.csd]

**Array Opcodes.**

*copya2ftab.csd* [examples/copya2ftab.csd]

*copyf2array.csd* [examples/copyf2array.csd]

*maxarray.csd* [examples/maxarray.csd]

*minarray.csd* [examples/minarray.csd]

*scalearray.csd* [examples/scalearray.csd]

*sumarray.csd* [examples/sumarray.csd]

**Miscellaneous.**

*directory.csd* [examples/directory.csd]

*fareyleni.csd* [examples/fareyleni.csd]

*modmatrix.csd* [examples/modmatrix.csd]

*pwd.csd* [examples/pwd.csd]

*select.csd* [examples/select.csd]

*system.csd* [examples/system.csd]

*farey7shuffled.csd* [examples/farey7shuffled.csd]

---

# Appendix B. Pitch Conversion

**Table B.1. Pitch Conversion**

Note	Hz	cpspch	MIDI
C-1	8.176	3.00	0
C#-1	8.662	3.01	1
D-1	9.177	3.02	2
D#-1	9.723	3.03	3
E-1	10.301	3.04	4
F-1	10.913	3.05	5
F#-1	11.562	3.06	6
G-1	12.250	3.07	7
G#-1	12.978	3.08	8
A-1	13.750	3.09	9
A#-1	14.568	3.10	10
B-1	15.434	3.11	11
C0	16.352	4.00	12
C#0	17.324	4.01	13
D0	18.354	4.02	14
D#0	19.445	4.03	15
E0	20.602	4.04	16
F0	21.827	4.05	17
F#0	23.125	4.06	18
G0	24.500	4.07	19
G#0	25.957	4.08	20
A0	27.500	4.09	21
A#0	29.135	4.10	22
B0	30.868	4.11	23
C1	32.703	5.00	24
C#1	34.648	5.01	25
D1	36.708	5.02	26
D#1	38.891	5.03	27
E1	41.203	5.04	28
F1	43.654	5.05	29
F#1	46.249	5.06	30
G1	48.999	5.07	31
G#1	51.913	5.08	32
A1	55.000	5.09	33

Pitch Conversion

<b>Note</b>	<b>Hz</b>	<b>cpspch</b>	<b>MIDI</b>
A#1	58.270	5.10	34
B1	61.735	5.11	35
C2	65.406	6.00	36
C#2	69.296	6.01	37
D2	73.416	6.02	38
D#2	77.782	6.03	39
E2	82.407	6.04	40
F2	87.307	6.05	41
F#2	92.499	6.06	42
G2	97.999	6.07	43
G#2	103.826	6.08	44
A2	110.000	6.09	45
A#2	116.541	6.10	46
B2	123.471	6.11	47
C3	130.813	7.00	48
C#3	138.591	7.01	49
D3	146.832	7.02	50
D#3	155.563	7.03	51
E3	164.814	7.04	52
F3	174.614	7.05	53
F#3	184.997	7.06	54
G3	195.998	7.07	55
G#3	207.652	7.08	56
A3	220.000	7.09	57
A#3	233.082	7.10	58
B3	246.942	7.11	59
C4	261.626	8.00	60
C#4	277.183	8.01	61
D4	293.665	8.02	62
D#4	311.127	8.03	63
E4	329.628	8.04	64
F4	349.228	8.05	65
F#4	369.994	8.06	66
G4	391.995	8.07	67
G#4	415.305	8.08	68
A4	440.000	8.09	69
A#4	466.164	8.10	70
B4	493.883	8.11	71

Pitch Conversion

<b>Note</b>	<b>Hz</b>	<b>cpspch</b>	<b>MIDI</b>
C5	523.251	9.00	72
C#5	554.365	9.01	73
D5	587.330	9.02	74
D#5	622.254	9.03	75
E5	659.255	9.04	76
F5	698.456	9.05	77
F#5	739.989	9.06	78
G5	783.991	9.07	79
G#5	830.609	9.08	80
A5	880.000	9.09	81
A#5	932.328	9.10	82
B5	987.767	9.11	83
C6	1046.502	10.00	84
C#6	1108.731	10.01	85
D6	1174.659	10.02	86
D#6	1244.508	10.03	87
E6	1318.510	10.04	88
F6	1396.913	10.05	89
F#6	1479.978	10.06	90
G6	1567.982	10.07	91
G#6	1661.219	10.08	92
A6	1760.000	10.09	93
A#6	1864.655	10.10	94
B6	1975.533	10.11	95
C7	2093.005	11.00	96
C#7	2217.461	11.01	97
D7	2349.318	11.02	98
D#7	2489.016	11.03	99
E7	2637.020	11.04	100
F7	2793.826	11.05	101
F#7	2959.955	11.06	102
G7	3135.963	11.07	103
G#7	3322.438	11.08	104
A7	3520.000	11.09	105
A#7	3729.310	11.10	106
B7	3951.066	11.11	107
C8	4186.009	12.00	108
C#8	4434.922	12.01	109

---

Pitch Conversion

---

<b>Note</b>	<b>Hz</b>	<b>cpspch</b>	<b>MIDI</b>
D8	4698.636	12.02	110
D#8	4978.032	12.03	111
E8	5274.041	12.04	112
F8	5587.652	12.05	113
F#8	5919.911	12.06	114
G8	6271.927	12.07	115
G#8	6644.875	12.08	116
A8	7040.000	12.09	117
A#8	7458.620	12.10	118
B8	7902.133	12.11	119
C9	8372.018	13.00	120
C#9	8869.844	13.01	121
D9	9397.273	13.02	122
D#9	9956.063	13.03	123
E9	10548.08	13.04	124
F9	11175.30	13.05	125
F#9	11839.82	13.06	126
G9	12543.85	13.07	127

---

# Appendix C. Sound Intensity Values

**Table C.1. Sound Intensity Values (for a 1000 Hz tone)**

<b>Dynamics</b>	<b>Intensity (W/m<sup>2</sup>)</b>	<b>Level (dB)</b>
pain	1	120
fff	10 <sup>-2</sup>	100
f	10 <sup>-4</sup>	80
p	10 <sup>-6</sup>	60
ppp	10 <sup>-8</sup>	40
threshold	10 <sup>-12</sup>	0

---

# Appendix D. Formant Values

**Table D.1. alto “a”**

Values	f1	f2	f3	f4	f5
freq (Hz)	800	1150	2800	3500	4950
amp (dB)	0	-4	-20	-36	-60
bw (Hz)	80	90	120	130	140

**Table D.2. alto “e”**

Values	f1	f2	f3	f4	f5
freq (Hz)	400	1600	2700	3300	4950
amp (dB)	0	-24	-30	-35	-60
bw (Hz)	60	80	120	150	200

**Table D.3. alto “i”**

Values	f1	f2	f3	f4	f5
freq (Hz)	350	1700	2700	3700	4950
amp (dB)	0	-20	-30	-36	-60
bw (Hz)	50	100	120	150	200

**Table D.4. alto “o”**

Values	f1	f2	f3	f4	f5
freq (Hz)	450	800	2830	3500	4950
amp (dB)	0	-9	-16	-28	-55
bw (Hz)	70	80	100	130	135

**Table D.5. alto “u”**

Values	f1	f2	f3	f4	f5
freq (Hz)	325	700	2530	3500	4950
amp (dB)	0	-12	-30	-40	-64
bw (Hz)	50	60	170	180	200

**Table D.6. bass “a”**

Values	f1	f2	f3	f4	f5
freq (Hz)	600	1040	2250	2450	2750
amp (dB)	0	-7	-9	-9	-20
bw (Hz)	60	70	110	120	130

**Table D.7. bass “e”**

Values	f1	f2	f3	f4	f5
freq (Hz)	400	1620	2400	2800	3100
amp (dB)	0	-12	-9	-12	-18
bw (Hz)	40	80	100	120	120

**Table D.8. bass “i”**

Values	f1	f2	f3	f4	f5
freq (Hz)	250	1750	2600	3050	3340
amp (dB)	0	-30	-16	-22	-28
bw (Hz)	60	90	100	120	120

**Table D.9. bass “o”**

Values	f1	f2	f3	f4	f5
freq (Hz)	400	750	2400	2600	2900
amp (dB)	0	-11	-21	-20	-40
bw (Hz)	40	80	100	120	120

**Table D.10. bass “u”**

Values	f1	f2	f3	f4	f5
freq (Hz)	350	600	2400	2675	2950
amp (dB)	0	-20	-32	-28	-36
bw (Hz)	40	80	100	120	120

**Table D.11. countertenor “a”**

Values	f1	f2	f3	f4	f5
freq (Hz)	660	1120	2750	3000	3350
amp (dB)	0	-6	-23	-24	-38
bw (Hz)	80	90	120	130	140

**Table D.12. countertenor “e”**

Values	f1	f2	f3	f4	f5
freq (Hz)	440	1800	2700	3000	3300
amp (dB)	0	-14	-18	-20	-20
bw (Hz)	70	80	100	120	120

**Table D.13. countertenor “i”**

Values	f1	f2	f3	f4	f5
freq (Hz)	270	1850	2900	3350	3590

Values	f1	f2	f3	f4	f5
amp (dB)	0	-24	-24	-36	-36
bw (Hz)	40	90	100	120	120

**Table D.14. countertenor “o”**

Values	f1	f2	f3	f4	f5
freq (Hz)	430	820	2700	3000	3300
amp (dB)	0	-10	-26	-22	-34
bw (Hz)	40	80	100	120	120

**Table D.15. countertenor “u”**

Values	f1	f2	f3	f4	f5
freq (Hz)	370	630	2750	3000	3400
amp (dB)	0	-20	-23	-30	-34
bw (Hz)	40	60	100	120	120

**Table D.16. soprano “a”**

Values	f1	f2	f3	f4	f5
freq (Hz)	800	1150	2900	3900	4950
amp (dB)	0	-6	-32	-20	-50
bw (Hz)	80	90	120	130	140

**Table D.17. soprano “e”**

Values	f1	f2	f3	f4	f5
freq (Hz)	350	2000	2800	3600	4950
amp (dB)	0	-20	-15	-40	-56
bw (Hz)	60	100	120	150	200

**Table D.18. soprano “i”**

Values	f1	f2	f3	f4	f5
freq (Hz)	270	2140	2950	3900	4950
amp (dB)	0	-12	-26	-26	-44
bw (Hz)	60	90	100	120	120

**Table D.19. soprano “o”**

Values	f1	f2	f3	f4	f5
freq (Hz)	450	800	2830	3800	4950
amp (dB)	0	-11	-22	-22	-50
bw (Hz)	40	80	100	120	120

**Table D.20. soprano “u”**

Values	f1	f2	f3	f4	f5
freq (Hz)	325	700	2700	3800	4950
amp (dB)	0	-16	-35	-40	-60
bw (Hz)	50	60	170	180	200

**Table D.21. tenor “a”**

Values	f1	f2	f3	f4	f5
freq (Hz)	650	1080	2650	2900	3250
amp (dB)	0	-6	-7	-8	-22
bw (Hz)	80	90	120	130	140

**Table D.22. tenor “e”**

Values	f1	f2	f3	f4	f5
freq (Hz)	400	1700	2600	3200	3580
amp (dB)	0	-14	-12	-14	-20
bw (Hz)	70	80	100	120	120

**Table D.23. tenor “i”**

Values	f1	f2	f3	f4	f5
freq (Hz)	290	1870	2800	3250	3540
amp (dB)	0	-15	-18	-20	-30
bw (Hz)	40	90	100	120	120

**Table D.24. tenor “o”**

Values	f1	f2	f3	f4	f5
freq (Hz)	400	800	2600	2800	3000
amp (dB)	0	-10	-12	-12	-26
bw (Hz)	70	80	100	130	135

**Table D.25. tenor “u”**

Values	f1	f2	f3	f4	f5
freq (Hz)	350	600	2700	2900	3300
amp (dB)	0	-20	-17	-14	-26
bw (Hz)	40	60	100	120	120

# Appendix E. Modal Frequency Ratios

## Contributed by Scott Lindroth

John Bower, a student of Scott Lindroth, compiled this list of modal frequencies for various objects and materials. Some modes work better than others, and most need to be in a particular frequency range to sound plausible. Caveat emptor.

In general, wooden objects will not sound "wooden" unless a stochastic component is present in the sound (try banded waveguides). Nonetheless, some of the wooden objects make wonderful metallic instruments as well.

This ratios can be useful together with opcodes like *mode* or *streson*.

**Table E.1. Modal Frequency Ratios**

Instrument	Modal Frequency Ratios
Dahina tabla	[1, 2.89, 4.95, 6.99, 8.01, 9.02]
Bayan tabla	[1, 2.0, 3.01, 4.01, 4.69, 5.63]
Red Cedar wood plate	[1, 1.47, 2.09, 2.56]
Redwood wood plate	[1, 1.47, 2.11, 2.57]
Douglas Fir wood plate	[1, 1.42, 2.11, 2.47]
uniform wooden bar	[1, 2.572, 4.644, 6.984, 9.723, 12]
uniform aluminum bar	[1, 2.756, 5.423, 8.988, 13.448, 18.680]
Xylophone	[1, 3.932, 9.538, 16.688, 24.566, 31.147]
Vibraphone 1	[1, 3.984, 10.668, 17.979, 23.679, 33.642]
Vibraphone 2	[1, 3.997, 9.469, 15.566, 20.863, 29.440]
Chalandi plates	([62, 107, 360, 460, 863] Hz +-2Hz) [1, 1.72581, 5.80645, 7.41935, 13.91935] ratios
tibetan bowl (180mm)	( [221, 614, 1145, 1804, 2577, 3456, 4419] Hz) 934g, 180mm [1, 2.77828, 5.18099, 8.16289, 11.66063, 15.63801, 19.99 ratios
tibetan bowl (152 mm)	([314, 836, 1519, 2360, 3341, 4462, 5696] Hz) 563g, 152mm [1, 2.66242, 4.83757, 7.51592, 10.64012, 14.21019, 18.14027] ratios
tibetan bowl (140 mm)	([528, 1460, 2704, 4122, 5694] Hz) 557g, 140mm [1, 2.76515, 5.12121, 7.80681, 10.78409] ratios
Wine Glass	[1, 2.32, 4.25, 6.63, 9.38]
small handbell	([1312.0, 1314.5, 2353.3, 2362.9, 3306.5, 3309.4, 3923.8, 3928.2, 4966.6, 4993.7, 5994.4, 6003.0, 6598.9, 6619.7, 7971.7, 7753.2, 8413.1, 8453.3, 9292.4, 9305.2, 9602.3, 9912.4] Hz)

Modal Frequency Ratios

Instrument	Modal Frequency Ratios
	[ 1, 1.0019054878049, 1.7936737804878, 1.8009908536585, 2.5201981707317, 2.5224085365854, 2.9907012195122, 2.9940548780488, 3.7855182926829, 3.8061737804878, 4.5689024390244, 4.5754573170732, 5.0296493902439, 5.0455030487805, 6.0759908536585, 5.9094512195122, 6.4124237804878, 6.4430640243902, 7.0826219512195, 7.0923780487805, 7.3188262195122, 7.5551829268293 ] ratios
spinel sphere with diameter of 3.6675mm	([977.25, 1003.16, 1390.13, 1414.93, 1432.84, 1465.34, 1748.48, 1834.20, 1919.90, 1933.64, 1987.20, 2096.48, 2107.10, 2202.08, 2238.40, 2280.10, 0 /*2290.53 calculated*/, 2400.88, 2435.85, 2507.80, 2546.30, 2608.55, 2652.35, 2691.70, 2708.00] Hz)  [ 1, 1.026513174725, 1.4224916858532, 1.4478690202098, 1.4661959580455, 1.499452545408, 1.7891839345101, 1.8768994627782, 1.9645945254541, 1.9786543873113, 2.0334612432847, 2.1452852391916, 2.1561524686621, 2.2533435661294, 2.2905090816065, 2.3331798413917, 0, 2.4567715528268, 2.4925556408289, 2.5661806088514, 2.6055768738808, 2.6692760296751, 2.7140956766436, 2.7543617293425, 2.7710411870043 ] ratios
pot lid	[ 1, 3.2, 6.23, 6.27, 9.92, 14.15] ratios

---

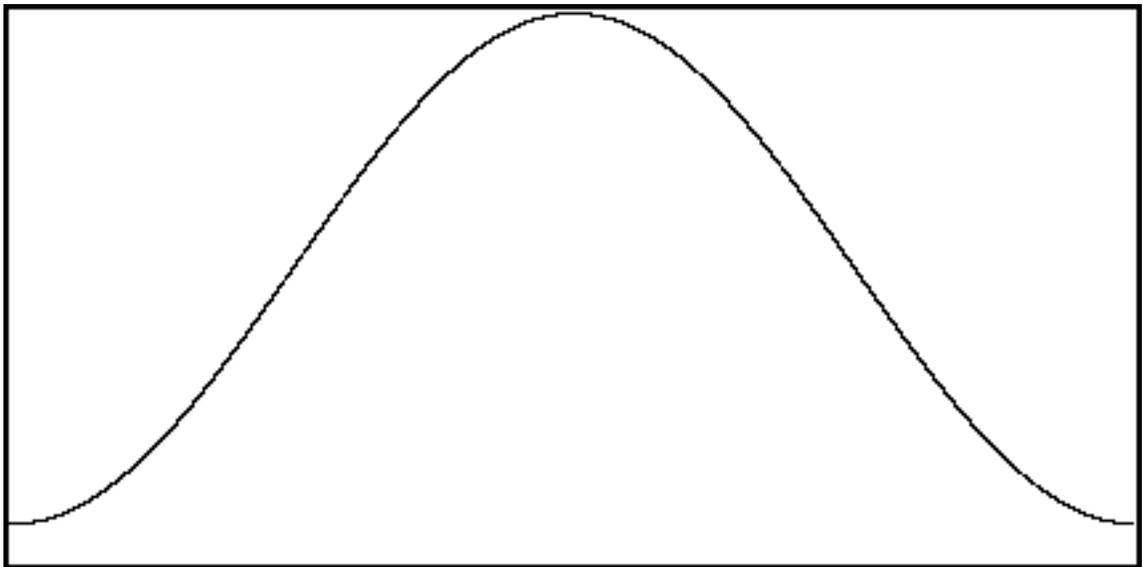
# Appendix F. Window Functions

Windowing functions are used for analysis, and as waveform envelopes, particularly in granular synthesis. Window functions are built in to some opcodes, but others require a function table to generate the window. *GEN20* is used for this purpose. The diagram of each window below, is accompanied by the `f` statement used to generate the it.

## Hamming.

### Example F.1. Hamming window function statement

```
f81 0 8192 20 1 1
```

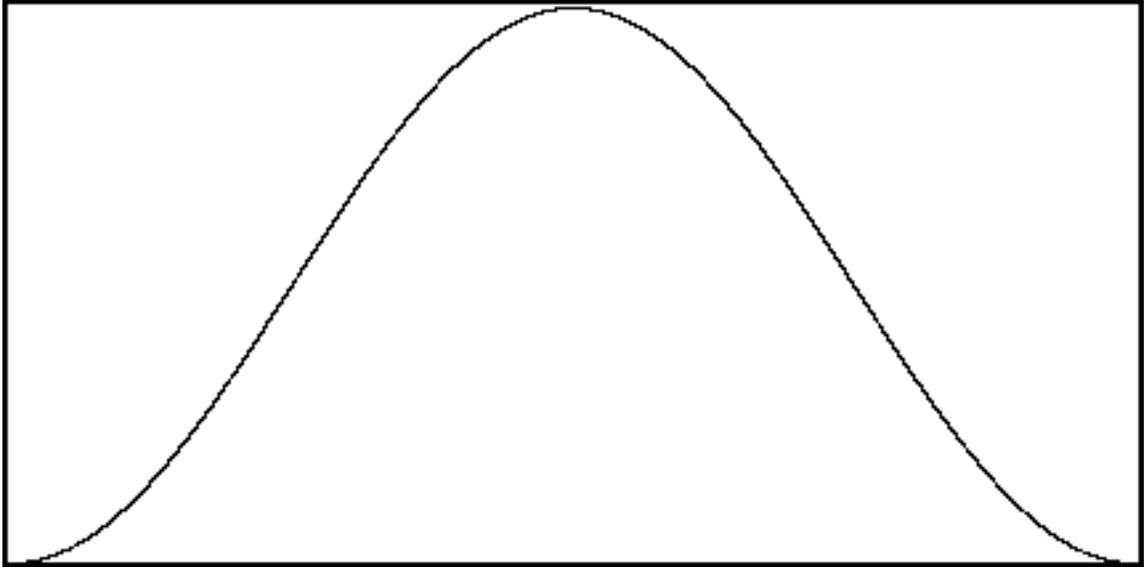


Hamming Window Function.

## Hanning.

### Example F.2. Hanning window function statement

```
f82 0 8192 20 2 1
```

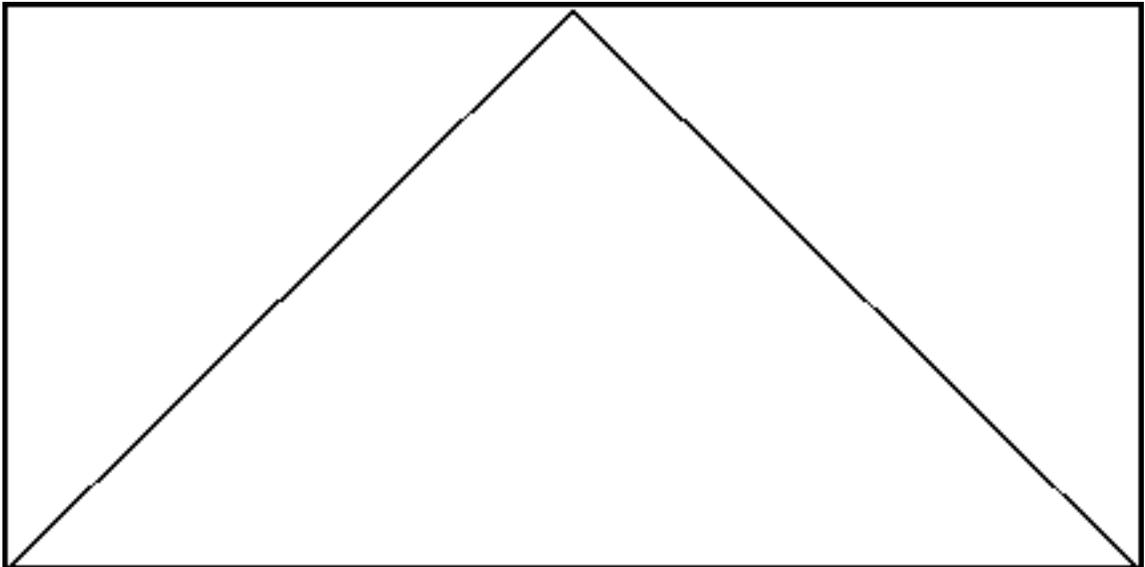


Hanning Window Function

**Bartlett.**

**Example F.3. Bartlett window function statement**

```
f83 0 8192 20 3 1
```

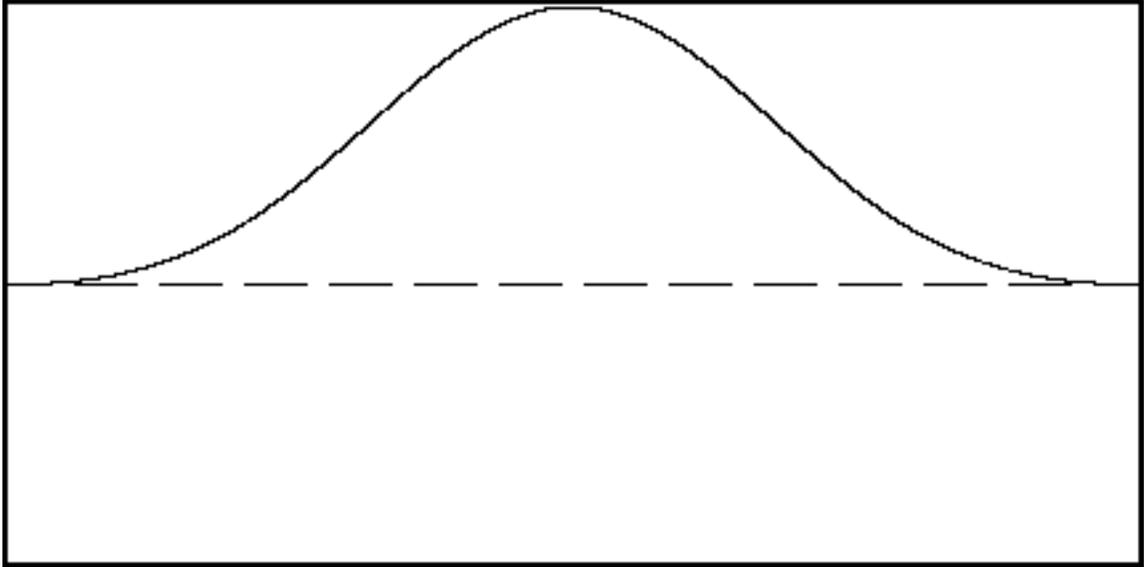


Bartlett Window Function

**Blackman.**

**Example F.4. Blackman window function statement**

```
f84 0 8192 20 4 1
```

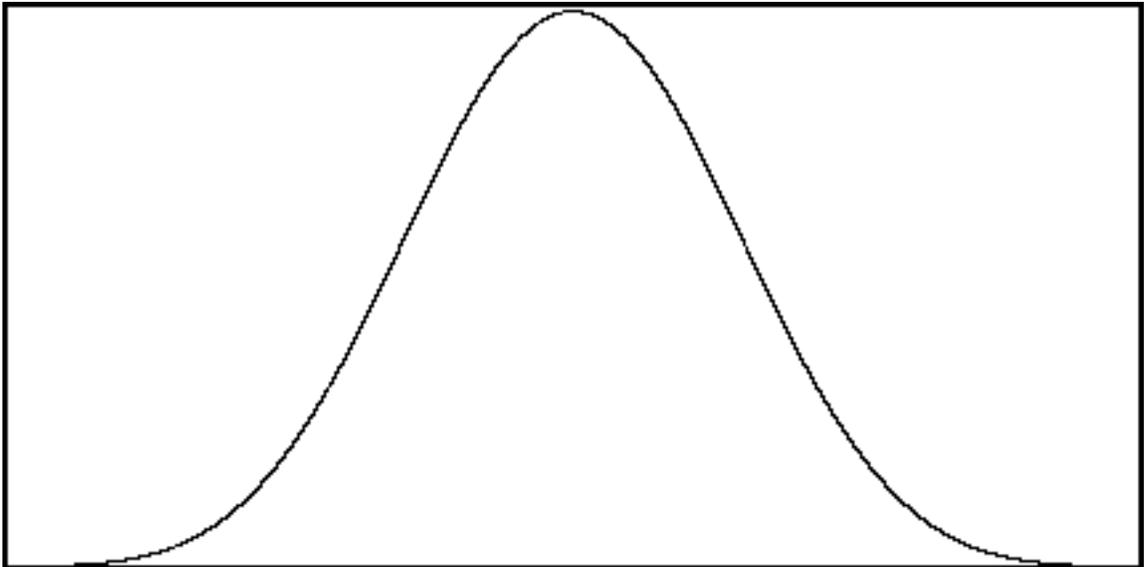


Blackman Window Function

**Blackman-Harris.**

**Example F.5. Blackman-Harris window function statement**

```
f85 0 8192 20 5 1
```

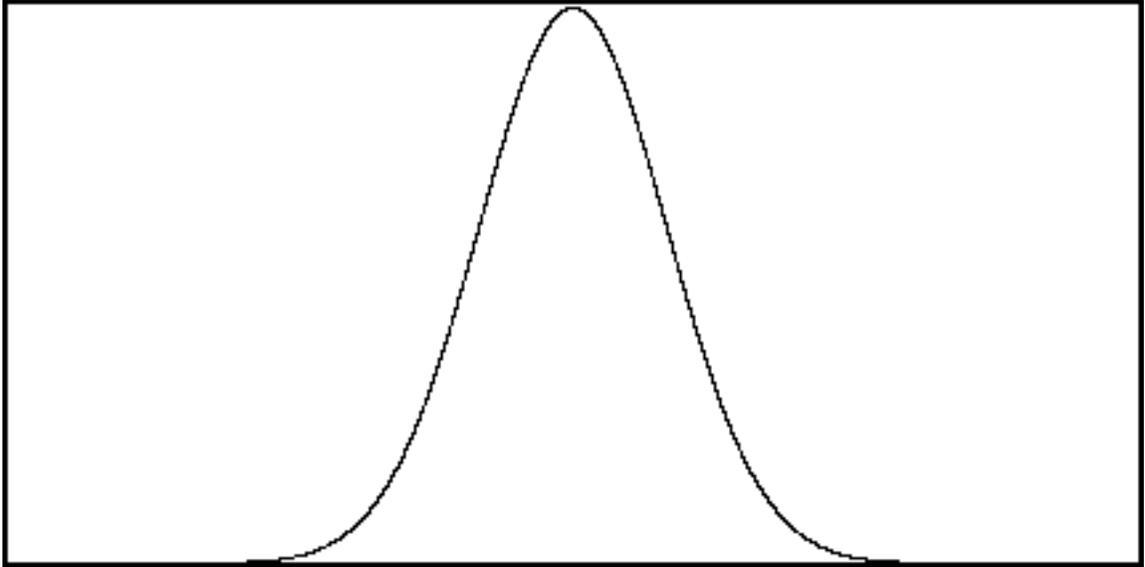


Blackman-Harris Window Function

**Gaussian.**

**Example F.6. Gaussian window function statement**

```
f86 0 8192 20 6 1
```



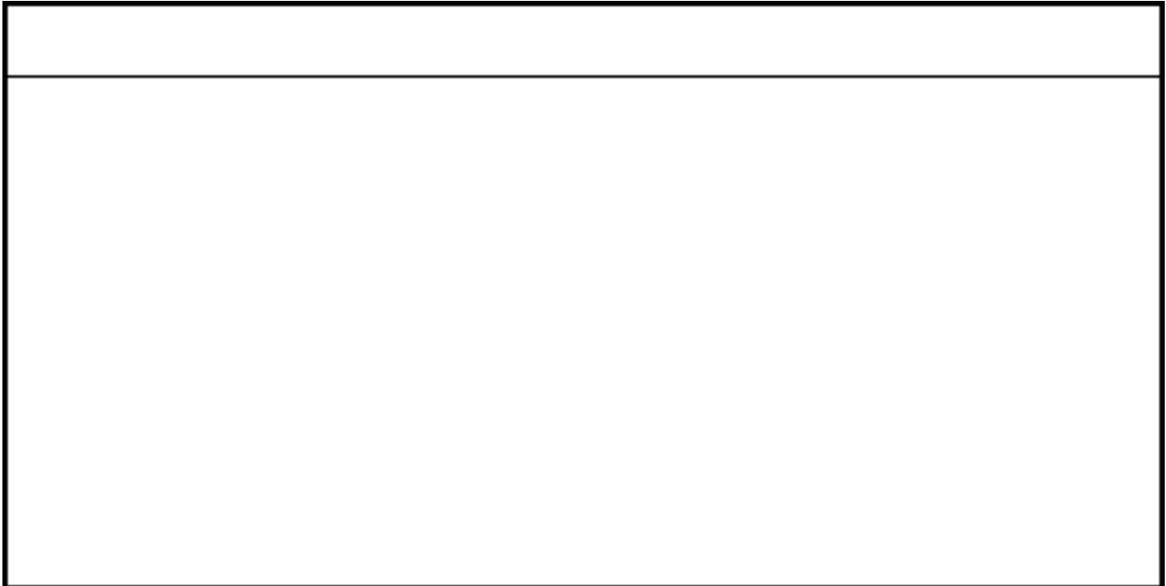
Gaussian Window Function

**Rectangle.**

**Example F.7. Rectangle window function statement**

```
f88 0 8192 -20 8 .1
```

*Note:* Vertical scale is exaggerated in this diagram.

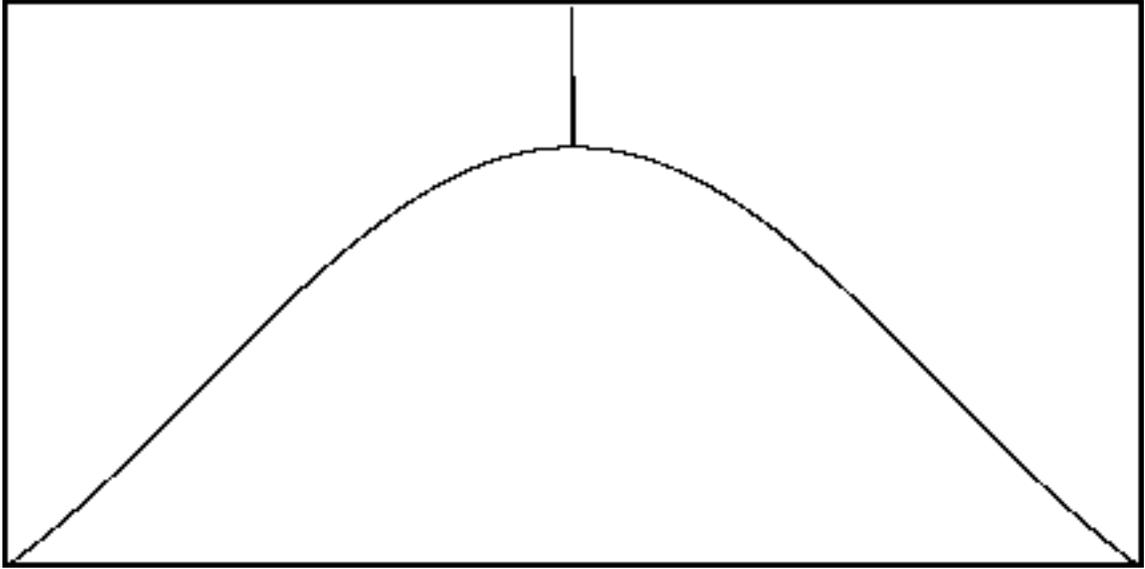


Rectangle Window Function

**Sync.**

**Example F.8. Sync window function statement**

```
f89 0 4096 -20 9 .75
```



Sinc Window Function

---

# Appendix G. SoundFont2 File Format

Beginning with Csound Version 4.07, *Csound supports the SoundFont2 sample file format*. SoundFont2 (or SF2) is a widespread standard which allows encoding banks of wavetable-based sounds into a binary file. In order to understand the usage of these opcodes, the user must have some knowledge of the SF2 format, so a brief description of this format follows.

The SF2 format is made by generator and modulator objects. All current Csound opcodes regarding SF2 support the generator function only.

There are several levels of generators having a hierarchical structure. The most basic kind of generator object is a sample. Samples may or may not be be looped, and are associated with a MIDI note number, called the base-key. When a sample is associated with a range of MIDI note numbers, a range of velocities, a transposition (coarse and fine tuning), a scale tuning, and a level scaling factor, the sample and its associations make up a “split.” A set of splits, together with a name, make up an “instrument.” When an instrument is associated with a key range, a velocity range, a level scaling factor, and a transposition, the instrument and its associations make up a “layer.” A set of layers, together with a name, makes up a “preset.” Presets are normally the final sound-generating structures ready for the user. They generate sound according to the settings of their lower-level components.

Both sample data and structure data is embedded in the same SF2 binary file. A single SF2 file can contain up to a maximum of 128 banks of 128 preset programs, for a total of 16384 presets in one SF2 file. The maximum number of layers, instruments, splits, and samples is not defined, and probably is only limited by the computer's memory.

---

# Appendix H. Csound Double (64-bit) vs. Float (32-bit)

Csound can be built to use 64-bit DOUBLES internally to do processing versus regular Csound's 32-bit FLOATS. This larger resolution for processing internally yields a much "cleaner" sound but at the expense of extended processing time. Because it does require much longer to process, Csound compiled for doubles is typically used after a work is finished for a final production run. If you are using csound for realtime output, you should use the 32-bit (float) version, which provides faster output. For offline rendering, you can use either, but for the final master, the 64-bit version will produce higher quality output.

The actual resolution should be the same as for the type of the audio sample variable. For "float" Csound, that is a 32-bit, single-precision floating point number. It has 24 bits of precision in the mantissa. For "double" Csound, that is a 64-bit, double-precision floating point number in the mantissa. It has 52 bits of precision. For each decimal digit, there is between 3 and 4 bits. So, there are 7 digits of precision for "float" and 16 digits of precision for "double."

For each multiplication or division, depending on whether the operands are integers, rational numbers, repeating decimals, or irrational decimals, there may or may not be rounding error in the result. If there is rounding error, it is at most 1 bit of lost precision per operation (in addition to any rounding in the binary representation of rational or real numbers).

For float samples, if the signal stays within the mantissa, the signal to noise ratio is 6.02 times 24, or 144 dB. At worst, each operation will create 6.02 dB of additional noise due to rounding error. Our ears have an effective dynamic range of 120 to 130 dB, but we like our music compressed to a dynamic range of AT MOST 60 dB (and usually much less, say 20 dB). That gives us  $(144 - 60) / 6.02 =$  about 10 worst-case operations before we could possibly hear any degradation. In practice, we have several times that many operations before we hear any degradation or noise.

For double samples, if the signal stays within the mantissa, the signal to noise ratio is 6.02 times 52, or 313 dB. At worst, each operation will create 6.02 dB of additional noise due to rounding error. That gives us  $(313 - 60) / 6.02 =$  about 42, in practice several times that, of operations before there is any audible degradation or noise.

But if you trace the number of arithmetic operations in typical Csound instruments or other software synthesizers, the very complex instruments definitely are pushing into the range of audible degradation on good monitors with float, so it is no surprise that in some cases blindfold, controlled ABX testing confirms *occasional* audible differences between music synthesized with "float" Csound versus the same music synthesized with "double" Csound. Equally, it is no surprise that there are easily audible differences between digital and analog implementations of the same synthesis algorithms.

There is a good deal more "digital signal processing headroom" with double Csound, and therefore it should be used for all music rendered for critical listening. The float version should only be used where its speed advantage of about 15% is critical for real-time performance.

## Notes On Using Csound built for double precision

1. hetro, PVOC-EX analysis and pval files generated for 32-bit Csound (float) will work with 64-bit Csound (double precision).
2. All other analysis formats can be generated in a way for cross-use between 32 and 64 bit versions and big-endian and little-endian architectures, if the -X option is used in generation. The older format in csound5 is still generatable but is not machins independant.

---

# Glossary

## G

### Guard Point

A guard point is the last position on a function table. If the length is, say 1024, the table will have 1024+1 (1025) points: the extra point is the guard point.

In any case, for a 1024-point table, the first point is index 0 and the last 1023; index 1024 is not really used)

The reason for a guard-point is that some opcodes interpolate to obtain a table value, in which case, when the table index is say, 1023.5, we need the value of the 1024 pos in order to interpolate.

There are two ways of filling this point (writing the value that goes in it):

1. Default way: by copying the value of the 1st point in the table
2. Extended Guard-Point: extending the contour of the table (continuing to calculate the table for one extra point)

In general the first mode is used for wrap-around applications, such as an oscillator (which loops continuously reading the table). The second use is for one-shot readouts, such as envelopes, where the last point needs to be interpolated correctly following the table contour (we are not looping back to the beginning of the table)